SHORT HORIZON LEARNING-BASED SPEED PREDICTION FOR ELECTRIC VEHICLES

A Dissertation
Presented to
The Academic Faculty

By

Peter W. Somers

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Mechanical Engineering

Georgia Institute of Technology

May 2020

SHORT HORIZON LEARNING-BASED SPEED PREDICTION FOR ELECTRIC VEHICLES

Approved by:

Prof. Bert Bras, Advisor
School of Mechanical Engineering
Georgia Institute of Technology

Prof. Andrei Fedorov
School of Mechanical Engineering
Georgia Institute of Technology

Prof. Dr.-Ing. Dr. h.c. Oliver Sawodny
Institute for System Dynamics
University of Stuttgart

Prof. Dr.-Ing. Cristina Tarín
Institute for System Dynamics
University of Stuttgart

Date Approved: January 11, 2000

ACKNOWLEDGEMENTS

I would like to first and foremost thank Dr. Oliver Sawodny and Dr. Paul Neitzel for providing me with the opportunity to participate in this Joint Master's program. In particular, many thanks to Dr. Sawodny for ensuring that my transition to Germany went as smoothly as possible and for organizing my working position at Daimler AG. Your efforts made my first trip abroad as completely stress-free and enjoyable as possible.

I must admit, I don't know how I would have fared in this venture without the constant advice and support of my thesis advisor at both Daimler and Universität Stuttgart, Bernhard Rolle. You, along with Melanie Heinz, made my time at Daimler a completely unforgettable experience.

I would also like to thank Dr. Bert Bras for supporting my thesis efforts from the Georgia Tech side and both Dr. Cristina Tarín and Dr. Andrei Fedorov for participating on my committee.

Special thanks also go to my friend Samuel Tovey for our frequent, fruitful discussions and mutual frustrations regarding neural networks that spawned much inspiration during my work. And also to Julius and Paul, my brother, for much support during a difficult recovery period that caused a slow start to this thesis work.

Thank you also to Julius, Sam, Nicole, and Paul for taking the time to proofread this thesis and provide helpful comments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Acronyms

EV	Electric Vehicle
SOC	State of Charge
ACC	Adaptive Cruise Control
MPC	Model Predictive Control
EGO	Vehicle of Interest
ANN	Artificial Neural Network
NN	(Artificial) Neural Network
FFNN	Feed Forward Neural Network
FC	Fully Connected
RNN	Recurrent Neural Network
LSTM	Long Short Term Memory
CNN	Convolutional Neural Network
FCN	Fully-Convolutional Network
TCN	Temporal Convolutional Network
BBTT	Back-Propagation Through Time
DAIN	Deep Adaptive Input Normalization
REST	Representational State Transfer
API	Application Programming Interface
GUI	Graphical User Interface
CAN	Controller Area Network
MSE	Mean Squared Error
MAE	Mean Absolute Error
CS	Constant Speed
CA	Constant Acceleration
TSNN	Temporal-Spatial Neural Network
TONN	Temporal-Only Neural Network

SUMMARY

The automotive industry is moving more to the development of electric vehicles to meet environmental and emissions restrictions. As a result, much work is being done to optimize the efficiency of these vehicles through the use of various control methods such as model predictive control. These efforts often rely on the knowledge of future vehicle speed, however, this information is difficult to predict beyond a trivially small horizon. This work proposes including route information with onboard vehicle data to make longer speed predictions. This is done through the use of a new B-spline prediction concept in conjunction with a custom temporal-spatial neural network (TSNN) structure. The B-Spline prediction method is demonstrated first on a simple identification task and then the TSNN is trained on test vehicle data combined with route information from HERE maps. The TSNN was successfully shown to benefit from inclusion of the route information and outperform simple existing prediction methods.

CHAPTER 1
INTRODUCTION

In recent years, machine learning techniques have become a larger and larger focus for many disciplines from medicine to engineering. In particular, artificial neural networks (ANNs) have captured the attention of many scientists due to their seemingly unlimited capability of learning unknown relations between different variables[1]. When given enough training data, this helps the users to create effective tools for advancing their fields, such as improved diagnoses in medicine[2]. The capabilities of neural networks has specifically not been lost on the automobile industry.

With autonomous vehicles and driver assistance systems becoming more and more prevalent as consumer desired features, the automobile industry is tasked with advancing these fields in particular. Many have found the most effective tools come from machine learning[3] because they allow for the extraction of abstract information from data sets. For example, ANNs have been used with tasks such as online traffic sign recognition[4], which is used for both autonomous and assistive systems.

Throughout the development of a vehicle, many tests are performed, namely endurance driving tests, to validate the design. Due to technological advancements in data recording and storage devices, more information is now collected from these in-development vehicles than before as all signals can now be recorded at all times. This data is then made available to engineers for post-processing and makes data-driven analysis approaches more attractive[5]. In addition, ANNs can be used with this data for modeling highly-nonlinear information such as driving characteristics of a human that previously was not very feasible.

The vehicle platforms manufacturers are developing for consumers are continually moving more towards fully electric vehicles (EVs) in order to meet rising restrictions on

emissions and fuel consumption[6]. With the move to EVs, a new consumer concern has arrived concerning the range these EVs can travel. To address this concern, a focus has turned to increasing the efficiency of operation of an EV so that the range of the vehicle is maximized.

A popular and effective way to increase the efficiencies of a vehicle is to focus on optimizing the control inputs regarding different aspects of the vehicle[7, 8, 9], but most importantly the vehicle speed[10, 11, 12]. This work will focus on providing a ANN method to aid in providing these optimization efforts with a crucial piece of information: future desired vehicle speed. This information is of particular interest to many Model Predictive Control (MPC) systems[13]. A short introduction of the current work being done towards vehicle speed prediction is provided in the next section.

1.1 Vehicle Speed Prediction

In order to implement MPC algorithms it is necessary to have a defined prediction horizon over which a variable, or multiple, is optimized. In most cases, when trying to minimize energy consumption for a vehicle, the desired longitudinal speed of a vehicle is needed at the end of the prediction horizon[14]. Most often a simple longitudinal vehicle model with a constant input is used to predict the vehicle speed a very short horizon into the future, however, these methods are not sufficient for longer prediction horizons due to human driver inputs and traffic conditions[15].

This work aims to make an accurate prediction of desired vehicle speed utilizing additional data beyond what a simple vehicle model is capable of including. As the prediction horizon increases, the influence of uncertainties becomes more prevalent and the inclusion of additional information such as upcoming traffic conditions and route information provided by commericial navigational services should improve the prediction accuracy. Unfortunately, this information comes from sources that are fundamentally different in nature. For example, one is temporal data provided by the vehicle sensors, e.g. lidar distance and

speed, and another is spatial data from services such as HERE, e.g. road curvature and stop signs. The driver behavior can be obtained from signals such as the brake and accelerator pedal positions and steering angle. ANNs provide a versatile tool for combining all this data and extracting the highly nonlinear underlying relationships in order to make longer predictions. These speed predictions can then be used in conjunction with an MPC controller, such as the Adaptive Cruise Control (ACC) proposed in [12], to increase efficiency. The work done in [12] also uses the route information, such as road slope, to aid in determining the appropriate inputs to efficiently reach, or in their case, maintain, the desired speed.

There are many other current works that propose the use of neural networks for use in vehicle speed prediction. However, most of these are used for general traffic speed predictions and not direct predictions for the vehicle of interest (EGO vehicle)[16, 17, 18, 19]. A thorough comparison of parametric methods (constant input model propagation and similar) with non-parametric prediction methods (Gaussian Mixture Regression and neural networks) was performed by Lefevre in [15] to investigate effectiveness for different predictions horizons up to 10 seconds. From this work, Lefevre found that the neural networks performed equal to or better than all the other methods, however, it is worth noting that this work was only done for highway cruising, and no route information was included, just the information of preceding vehicles.

A similar comparison was done in [20], where it was also found that a NN out-performed an auto-regressive approach when predicting vehicle speed at specific prediction intervals up to 10 seconds. Rezaei also presented a comparison of different works in [14] and proposed an auto-regressive approach for vehicle speed predictions for the same horizon length. Rezaei included route events as inputs using the distance from current vehicle position. The approach suffers in city driving from non-known acceleration rates caused by driver characteristics that are not modeled.

One of the works initially influencing the efforts here within is that of Park in [21],

where multiple simple neural networks are used to learn driving behavior under separate driving conditions that were determined from route information. The networks successfully were able to extract vehicle behavior from the training set. However, only a single route was driven multiple times and used for both training and validation sets. It is unknown if the method generalizes well to include unseen routes and since the predictions are based on preset distances along the route, it would be difficult to implement the results directly for uses such as MPC.

In [11] a neural network was used in an urban environment to predict the preceding vehicle's speed up to a 3 second prediction horizon in order to develop an energy optimized speed profile for the EGO vehicle when approaching a traffic signal. The work required online knowledge of the traffic light status and was only evaluated on a single short route segment including 4 intersections.

Overall, it is only recently that route information has begun to be included in both vehicle and traffic speed analysis since it is now much more obtainable information with services such as HERE maps, whether or not the works use neural networks as mentioned or more parametric approaches[22]. It is also apparent that most, if not all, EGO speed prediction methods developed in current works utilize only a single given route driven multiple times for both training and validation and do not also validate the proposed methods on an unseen route. This work will extend the validation efforts to include driving a training route in the reverse direction that is not seen during training.

1.2 Thesis Structure

This work proposes a new method using neural networks to predict a continuous vehicle speed profile with the inclusion of route information. The prediction horizon will be long enough to be influenced by driver actions. The structure of this thesis is organized in the following way.

First, an introduction to neural networks is provided in Chapter 2, including various

network topologies and the method with which these neural networks are trained. Specific ANN structures are introduced that will be used within the scope of this work.

Following this, a novel ANN prediction method is proposed with the use of B-splines in Chapter 3. This method is outlined along with a brief introduction to the machine learning tool that is used to implement it. This tool is also used for all of the ANN implementations presented in this work. An example usage of the B-spline method is then investigated in an effort to model the output torque given from an EV motor system.

The B-spline prediction method is applied in Chapter 4 to the short horizon speed prediction problem as already introduced. A comparison is given between the inclusion and omission of route information for the proposed ANN structure and the prediction results are compared with existing simple prediction methods as used in [15].

Finally, an overview of the presented work is given and suggestions for future work are put forth.

CHAPTER 2
ARTIFICIAL NEURAL NETWORKS

Artificial neural networks are an attempt to recreate, in software, the way that the human brain reacts given a certain situation. The basic idea behind an ANN is the linking of several simple functions (neurons) together in different configurations in order to form a single comprehensive function that is able to map a set of inputs to the desired output. There is an almost limitless number of configurations these neurons can be placed in. Through the rest of this work, the ”artificial” specification is dropped and just neural network (NN) is used for brevity. This chapter will provide an introduction to how these networks function and learn and includes an overview of some of the most commonly used network formulations that are also used in this work.

2.1 Feed Forward Neural Network

The Feed Forward Neural Network (FFNN) is the backbone of all subsequent neural net structures. It is also the simplest and most commonly used neural net. The more complicated structures, some of which are described in the next sections, ultimately use the same inter-layer connection scheme as is explained for the FFNN.

To build the FFNN it is first necessary to introduce the concept of a neuron, or more commonly known in use, a *unit*. The idea of the artificial neuron was first introduced in 1943 by Pitts and McCulloch[23]. A neuron is the first step in attempting to convert the inputs to the desired output. Pitts and McCulloch's implementation was very limited and relied on mostly binary operations and thresholds. To increase the capability, Rosenblatt introduced the perceptron, which used a more flexible threshold and allowed him to successfully classify inputs based on two outputs[24]. This led to the current implementation that is simply a function $\psi(z) \in \mathbb{R}$ where z is either a direct input to a neural network or a

Table 2.1: Neuron Activation Functions.

	$\psi(z)$	$\frac{d\psi}{dz}$	Usage
Linear	z	1	input layer, output layer
Sigmoid	$\frac{1}{1+e^{-z}}$	$\psi(z)(1-\psi(z))$	gates in LSTM cell (Ch. 2.3)
ReLU	0 for $z<0$ z for $z\geq 0$	0 for $z<0$ 1 for $z\geq 0$	Generally used. Particularly for CNN (Ch. 2.4)
tanh	$tanh(z)=\frac{(e^z-e^{-z})}{(e^z+e^{-z})}$	$1-\psi(z)^2$	internal activations of LSTM (Ch. 2.3)

weighted collection of inputs from upstream neurons. This function is commonly known as the activation function for the neuron. It is given this name because this function controls how effective the neuron's output is on the rest of the network. For example, if the output is near or equivalent to 0, the neuron is considered to be not active. Common activation functions that are used in this work are listed in Table 2.1.

While these neurons can be organized into almost any desirable configuration, there is a generally accepted structure which amounts to organizing groups of neurons into *layers*. These layers are stacked in series to form the general FFNN structure shown in Figure 2.1. Here z_i is the input to a layer and is equal to the weighted output of the previous layer x_i. Note that at the input layer $x_1 = z_1$, as the input has no additional weights applied. The notation $a_{\mathrm{i}}^{\mathrm{j}}$ is introduced to represent the activation value for the j-th neuron in the i-th layer. For the last layer, the activations are the output, which is highlighted by the notation $o_{\mathrm{j}} = a_{\mathrm{i}}^{\mathrm{j}}$. In further sections, only the o_{j} notation is used as intermediate activations are no longer of the focus. In future chapters, layers set up as described here will be referred to as fully connected (FC) layers.

With this structure the input z to the activation function for the j-th neuron in a layer is the weighted output of its N connections to the previous layer. For a fully connected layer

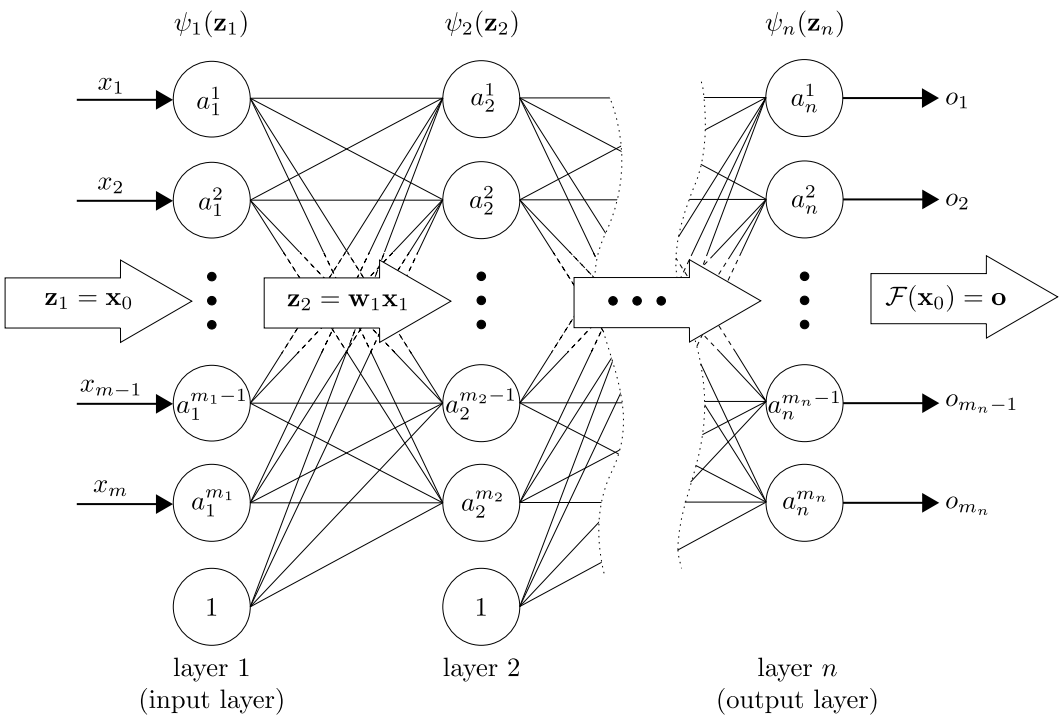


Figure 2.1: Feed Forward Neural Network Structure.

$N = m_{\mathrm{i}-1}$, where m_{i} is the number of neurons in layer i. This gives

$$z^{\mathrm{j}} = \sum_{\mathrm{k}=1}^{N} w_{\mathrm{k}} x_{\mathrm{k}} + b^{\mathrm{j}} \tag{2.1}$$

$$a_{\mathrm{i}}^{\mathrm{j}} = \psi_{\mathrm{i}}(z_{\mathrm{j}}), \tag{2.2}$$

where w_{k} is a weighting coefficient applied to the corresponding output x_{k} of the previous layer and b_{i} is the neuron's bias value. By using the same activation function for each neuron in a single layer, these equations are more commonly organized by layer such that the vectorial output of layer i, $\mathbf{x}_{\mathrm{i}} \in \mathbb{R}^{m_{\mathrm{i}}}$, is given by the following equations:

$$\mathbf{z}_{\mathrm{i}} = \mathbf{w}_{\mathrm{i}-1}\mathbf{x}_{\mathrm{i}-1} + \mathbf{b}_{\mathrm{i}} \tag{2.3}$$

$$\mathbf{x}_\mathrm{i} = \psi_\mathrm{i}(\mathbf{z}_\mathrm{i}). \tag{2.4}$$

Here $\psi_\mathrm{i}(\cdot)$ is an element-wise implementation of the chosen activation function for layer i. The bias vector $b_i \in \mathbb{R}$ is implemented in practice by just appending a unit with a constant output value of 1 to the previous layer's output vector $\mathbf{x}_{\mathrm{i}-1}$, therefore increasing the dimension by 1, and then applying the weight matrix $\mathbf{w}_{\mathrm{i}-1}$ accounting for the increase in size. This can be seen in Figure 2.1. This Feed Forward (FF) formulation provides a compact recursive computation of the output of layer n (taken as the output layer $\mathbf{o}$) that is given by Algorithm 1.

Algorithm 1 Feed Forward

$\mathbf{z}_1 \leftarrow \mathbf{x}_0$
$\mathbf{x}_1 \leftarrow \psi_1(\mathbf{z}_1)$
for $i \leftarrow 2$ to n **do** ▷ input layer does not contain trainable weights
    $\mathbf{z}_i \leftarrow \mathbf{w}_{i-1}\mathbf{x}_{i-1} + \mathbf{b}_i$
    $\mathbf{x}_i \leftarrow \psi_i(\mathbf{z}_i)$
return $\mathbf{o} \leftarrow \mathbf{x}_n$

2.1.1 Back-Propagation

Now that the structure for the FFNN has been outlined, the next task is to show why it is set-up this way and that will be done by showing how its learning mechanism works. The learning for a FFNN, or any neural net for that matter, is done by gradually adjusting the neuron weights to determine an optimal set of weights, $\mathbf{w}_\mathrm{i}$ for the neurons in each layer. The process that was developed to accomplish this is known as *back-propagation*, introduced by Werbos in his PhD thesis in 1974[25].

Moving forward, it is helpful to remember that the goal of the network function $\mathcal{F} : \mathbb{R}^m \to \mathbb{R}^p$ is to become the equivalent of some unknown function $f : \mathbb{R}^m \to \mathbb{R}^p$ that maps the m inputs to the p outputs. This goal can be used to determine how the weights of the network should be changed by introducing an error function that relates $\mathcal{F}$ and f. Back-propagation uses the method of gradient descent and this error function in order to find how

each weight should be adjusted. The error value of a NN is also known in practice as the *loss* and these two terms will be used interchangeably throughout this work.

The choice of this error function $E : R^l \to R$, where l is the number of decion variables, i.e. the numnber of neuron weights in the network, is very critical to the learning of the network as it describes how well the network function $\mathcal{F}$ resembles the unknown relation f. The error function must be chosen such that it is continuously differentiable with respect to the output values and the reason for this will be made clear further on. Now, the following standard error function E may be introduced given a set of p training samples $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_p, \mathbf{y}_p)\}$:

$$E = \frac{1}{2} \sum_{\mathrm{i}=1}^{p} \|\mathbf{o}_\mathrm{i} - \mathbf{y}_\mathrm{i}\|^2, \tag{2.5}$$

where

$$\mathbf{o}_\mathrm{i} = \mathcal{F}(\mathbf{x}_\mathrm{i}). \tag{2.6}$$

The only variables not held constant at the evaluation of E are the neuron weights $\mathbf{w}$ since the samples are fixed values, which leads to the gradient of E being given by

$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_l} \right)^\top. \tag{2.7}$$

By choosing all of the activation functions for each layer to be continuous and differentiable it is possible to calculate this gradient for all sample values. The weights are then updated with

$$\nabla w_\mathrm{i} = -\gamma \frac{\partial E}{\partial w_\mathrm{i}} \quad \text{for} \quad \mathrm{i} = 1, \ldots, l, \tag{2.8}$$

where $\gamma \in \mathbb{R}$ is the learning rate, a hyper parameter, that defines the step length for the gradient descent step. This is a very important parameter as it controls how fast the weights will converge to the optimal solution. If this value is very small, it may take very many iterations to converge and is at a higher risk of falling into a local minimum. If too large, the weights may have difficulty converging at all. For more information regarding gradient

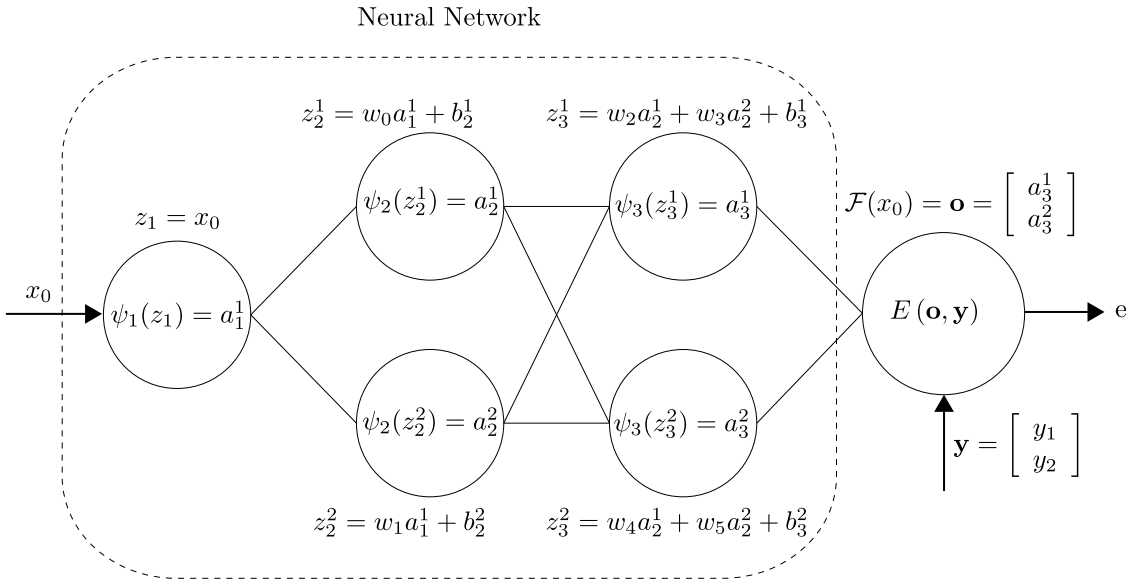


Figure 2.2: Example FFNN with loss.

descent and the effect of learning rate size on convergence, the reader is referred to [26].

Now that the requirements for back-propagation have been laid out, it is necessary to show how the value $\frac{\partial E}{\partial w_i}$ is efficiently calculated for each weight w_i. To do this, a sample network with two hidden layers is introduced that will be used to show how the weight adjustments are calculated and simultaneously the more general notation for a layered network is introduced. Also, since the training of the network is dependent on the chosen error function E, it is added to the sample network shown in Figure 2.2.

First, a single training sample (p=1) is assumed and with two output neurons in layer $n = 3$ the error becomes

$$e = E(\mathbf{o}, \mathbf{y}) = \frac{1}{2}(a_3^1 - y_1)^2 + \frac{1}{2}(a_3^2 - y_2)^2. \tag{2.9}$$

Here, a_3^j is the activation value of the j-th neuron in the output layer $n = 3$ and y_i is the corresponding training sample output. The concatenation of the activation values in the output layer represent the output for the entire neural network function $\mathcal{F}$. To demonstrate the derivation of the partial derivatives, the derivatives for the weights w_0 and w_2 will be calculated beginning with w_2.

Partial w.r.t. w_2

Using just a simple application of the chain rule from calculus and noting that the activation of neuron j in layer i is given by that layer's activation function $\psi_{\mathrm{i}}(z_{\mathrm{i}}^{\mathrm{j}})$, the partial derivative of the error function is given by

$$\frac{\partial E}{\partial w_2} = \left(a_3^1 - y_1\right) \frac{\partial \psi_3}{\partial w_2}(z_3^1) + \left(a_3^2 - y_2\right) \underbrace{\frac{\partial \psi_3}{\partial w_2}(z_3^2)}_{=0}. \tag{2.10}$$

The omission of replacing the notation a_3^{j} with that of the activation function is done purposely and will become clear further on. Also, the partial derivative of the activation function of the second output neuron is 0, because it is not dependent on w_2 as seen in Figure 2.2. Also, since the training sample is a constant value that cannot be changed, y_{j} drops out once the chain rule is applied. At this point a new variable is introduced to represent the error at each neuron, $\delta_{n+1}^{\mathrm{j}} = a_3^{\mathrm{j}} - y_{\mathrm{j}}$. Here the subscript is the layer and the superscript is the neuron. The error function itself is treated as an additional layer to allow for the $n + 1$ notation and (2.10) then becomes

$$\frac{\partial E}{\partial w_2} = \delta_{n+1}^1 \frac{\partial \psi_3}{\partial w_2}(z_3^1). \tag{2.11}$$

When expanding further with the chain rule using $z_3^1 = w_2 a_2^1 + w_3 a_2^2 + b_3^1$, the final result is obtained.

$$\frac{\partial E}{\partial w_2} = \underbrace{\delta_{n+1}^1 \psi_3'(z_3^1)}_{\delta_3^1} a_2^1 \tag{2.12}$$

Partial w.r.t. w_0

The partial derivative of the error function with respect to w_0 is given as

$$\frac{\partial E}{\partial w_0} = \left(a_3^1 - y_1\right) \frac{\partial \psi_3}{\partial w_0}(z_3^1) + \left(a_3^2 - y_2\right) \frac{\partial \psi_3}{\partial w_0}(z_3^2). \tag{2.13}$$

Note that the second term does not reduce to 0 as it did before. Following the same process as before the following is obtained:

$$\frac{\partial E}{\partial w_0} = \delta_3^1 w_2 \frac{\partial \psi_2}{\partial w_0}(z_2^1) + \delta_3^2 w_4 \frac{\partial \psi_2}{\partial w_0}(z_2^1) + \underbrace{\delta_3^1 w_3 \frac{\partial \psi_2}{\partial w_0}(z_2^2) + \delta_3^2 w_5 \frac{\partial \psi_2}{\partial w_0}(z_2^2)}_{=0}. \tag{2.14}$$

Note again, how the terms including $a_2^2 = \psi_2(z_2^2)$ vanish as they are not dependent on w_0. It is necessary now to apply the chain rule one more time to obtain the final expression.

$$\frac{\partial E}{\partial w_0} = \delta_3^1 w_2 \psi_2'(z_2^1) a_1^1 + \delta_3^2 w_4 \psi_2'(z_2^1) a_1^1 \tag{2.15}$$

This result can be re-arranged to the same form as the expression obtained for w_2. As before, everything except for the preceding neuron activation will collectively be considered in the *back-propagated error* for the neuron.

$$\frac{\partial E}{\partial w_0} = \underbrace{\left[\delta_3^1 w_2 + \delta_3^2 w_4\right] \psi_2'(z_2^1)}_{\textit{back-propagated error } \delta_2^1} a_1^1 \tag{2.16}$$

Generalized Formulation

Using the results obtained from the partial derivatives with respect to w_2 and w_0 and comparing to Figure 2.2, a pattern can be seen that takes the following form for a weight w_l that occurs in the j-th neuron of the i-th layer.

$$\frac{\partial E}{\partial w_l} = a_{\mathrm{i}-1}^* \psi_{\mathrm{i}}'(z_{\mathrm{i}}^{\mathrm{j}}) \sum_{\mathrm{k}=1}^{N} \delta_{\mathrm{i}+1}^{\mathrm{k}} w_{\mathrm{i}}^{\mathrm{k}}[\mathrm{j}] \tag{2.17}$$

Here, $*$ indicates the respective activation value from the previous layer neuron and is dependent on network structure. N is the number of neurons connected to the output of the current neuron and $w_{\mathrm{i}+1}^{\mathrm{k}}[\mathrm{j}]$ indicates the j-th weight of the k-th neuron connected in the next layer. This formulation is implemented in practice in layer form with Algorithm 2

assuming fully connected layers for an entire training operation.

Algorithm 2 Training Step with Back Propagation

$\mathbf{z}_1 \leftarrow \mathbf{x}_0$ ▷ feed forward operation ...
$\mathbf{x}_1 \leftarrow \psi_1(\mathbf{z}_1)$
for $i \leftarrow 2$ to n **do**
    $\mathbf{z}_i \leftarrow \mathbf{w}_{i-1}\mathbf{x}_{i-1} + \mathbf{b}_i$
    $\mathbf{x}_i \leftarrow \psi_i(\mathbf{z}_i)$
    Store $\psi_i' \leftarrow \frac{d\psi_i}{\delta \mathbf{z}_i}$, $\mathbf{x}_i$ ▷ additional step to FF
$\mathbf{o} \leftarrow \mathbf{x}_n$ ▷ back-propagation operation ...
$e \leftarrow E(\mathbf{o}, \mathbf{y})$
$\boldsymbol{\delta}_n \leftarrow \psi_n' \frac{de}{d\mathbf{o}}$
for $i \leftarrow n-1$ down to 1 **do**
    $\boldsymbol{\delta}_i = \boldsymbol{\delta}_{i+1}^\top \mathbf{w}_i$
    $\frac{\partial E}{\partial \mathbf{w}_i} = \mathbf{x}_{i-1}\psi_i'^\top \boldsymbol{\delta}_i$
    $\mathbf{w}_i \leftarrow \mathbf{w}_i - \gamma \frac{\partial E}{\partial \mathbf{w}_i}$

With everything provided so far a full neural network can be built and trained. However, the structure described thus far is incapable of learning dependencies between inputted values themselves (i.e. time series data) and so an additional method of recursively connecting a neuron to itself was proposed.

2.2 Recurrent Neural Network

Recurrent Neural Networks (RNNs), as they are known today, are based on the findings of [27] in which Rumelhart proposed an iterative neural network structure that allowed for the network to determine its own internal structure. This is done through the use of *hidden states* between the input and output. RNNs have become the leading structure for prediction tasks. The reason for this is that the hidden states are updated based on the effects of previous data and not just the current input. Naturally, this provides a useful tool for learning sequences.

The iterative structure proposed by Rumelhart can be seen in Figure 2.3 using 3 iterations of a 3-input network with the RNN representation on the left and the equivalent fully connected FFNN on the right. The key difference between this FFNN representation and a

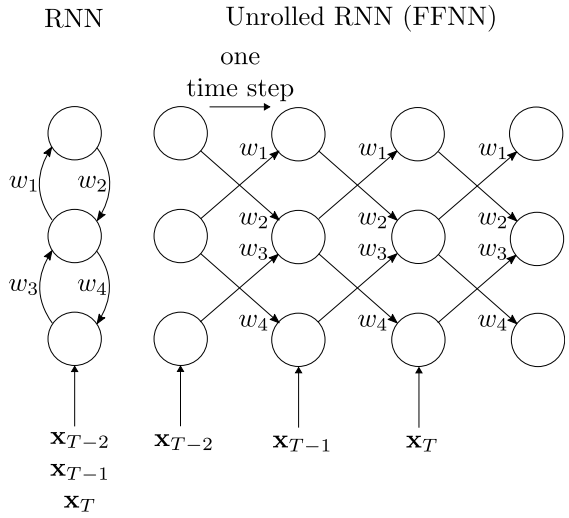


Figure 2.3: Equivalent Rolled Out RNN, cf. with [27].

"normal" FFNN as introduced in Section 2.1 is the shared weights across each layer. This FFNN representation may be referred to as a rolled-out RNN because it is a serial representation of the cyclical recurrent connections. A powerful feature of the RNN is that they do not have a limit to the length of the input vector. One may continue feeding time samples into the network and it will just keep its history by using the internal weights to add the previous hidden state to the current input. This is useful for non-fixed length data such as sentences in a language. However, this extensive history access can lead to issues that will be discussed further on.

The RNN structure outlined in Figure 2.3 can be considered a *cell*, and is treated similarly to a neuron in the aforementioned neural network structures. This general cell structure is shown in Figure 2.4 and also includes an output activation, ψ_o. What makes the RNN cell special is that it is repeated to encompass inputs and outputs across multiple time steps. The incoming connections for one copy are linked to the previous RNN cell output as well as the current input. Figure 2.4 shows a representation of the RNN structure with one time step and a single neuron, or unit, hidden state. The inputs for the internal hidden

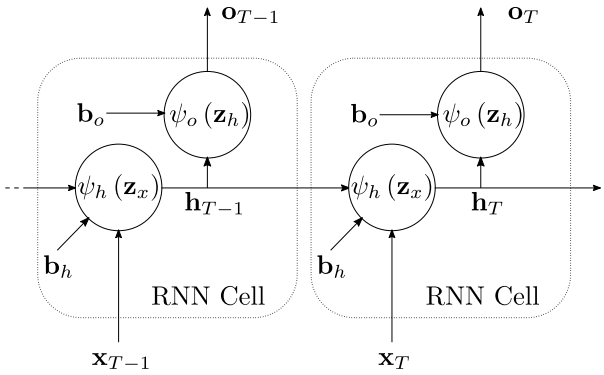


Figure 2.4: RNN Cells.

state activation, ψ_h, and the output activation, ψ_o, are

$$\mathbf{z}_x = \mathbf{w}_x \mathbf{x}_T + \mathbf{w}_h \mathbf{h}_{T-1} + \mathbf{b}_h \tag{2.18}$$

$$\mathbf{z}_h = \mathbf{w}_o \mathbf{h}_T + \mathbf{b}_o. \tag{2.19}$$

Here, T is the current time step, $\mathbf{h}$ is the hidden state, and $\mathbf{b}$ is the bias for the corresponding nonlinear activation function. Note that the hidden state element-wise activation, $\psi_h(\mathbf{z}_h)$, is typically extended to include many more hidden units with their own respective weights (i.e. it is viewed as a single layer with multiple neurons as in a FFNN) and are also time-invariant as is shown. This extension of the hidden state dimension allows the RNN to learn deeper relationships through time. It is also worth noting that the output activation shown here is simply a typical FFNN layer used to adapt the hidden state to the desired output dimension and can be moved outside of the cell if desired for a cleaner representation as it is not truly part of the recursive nature of the cell.

Through the use of the nonlinear activation functions and similar structure as a FFNN, it is possible to use back propagation for the weight updates. However, the algorithm introduced previously will not work directly with the RNN structure due to its dependence on time. To overcome this limitation, a slightly modified version known as *back-propagation*

through time (BBTT) was introduced.

The BBTT method is briefly mentioned by Rumelhart in [27], however, the first published and thorough explanation is given in [28] using FORTRAN code to explain the algorithms. A simplified, and preferred, pseudocode algorithm to calculate the gradient using BBTT is provided in [29], which is adapted to the notation used in this work in Algorithm 3. The adjusted standard loss function is now a sum of the losses for each time step

$$L\left(o_t; y_t\right) = \frac{1}{2}\sum_{t=1}^{T}\|\mathbf{o}_t - \mathbf{y}_t\|^2 \tag{2.20}$$

The back propagated errors for the output neurons at time t is given by $\delta_{o,t}$. The persisted error of the recurrent neurons is given by $\delta_{o+h,t}$ and $\delta_{x,t}$ is the error due to the input at time t.

Algorithm 3 Back Propagation Through Time

1: **Initialize:**
   $\boldsymbol{\delta}_{o+h,T} \leftarrow 0$ ▷ initialize persisted error
   $d\mathbf{w}_h, d\mathbf{w}_x, d\mathbf{w}_o, d\mathbf{b}_o, d\mathbf{b}_h \leftarrow 0$ ▷ derivatives of the weights to update
2: **for** $t \leftarrow T$ down to 1 **do**
3: $\boldsymbol{\delta}_{o,t} \leftarrow \psi_o{}'(\mathbf{z}_{h,t}) \cdot \partial L\left(\mathbf{o}_t; \mathbf{y}_t\right)/\partial \mathbf{o}_t$ ▷ error at output for current time step
4: $d\mathbf{b}_o \leftarrow d\mathbf{b}_o + \boldsymbol{\delta}_{o,t}$ ▷ change in output bias is direct accumulation of error
5: $d\mathbf{w}_o \leftarrow d\mathbf{w}_o + \boldsymbol{\delta}_{o,t}\mathbf{h}_t^\top$ ▷ add contribution of back-propagated error at current time
6: $\boldsymbol{\delta}_{o+h,t} \leftarrow \boldsymbol{\delta}_{o+h,t} + \mathbf{w}_o^\top \boldsymbol{\delta}_{o,t}$ ▷ update persisted error
7: $\boldsymbol{\delta}_{x,t} \leftarrow \psi_h{}'(\mathbf{z}_{x,t}) \cdot \boldsymbol{\delta}_{o+h,t}$ ▷ back propagated error for input weights
8: $d\mathbf{w}_x \leftarrow d\mathbf{w}_x + \boldsymbol{\delta}_{x,t}\mathbf{x}_t^\top$ ▷ add contribution to input weights
9: $d\mathbf{b}_h \leftarrow d\mathbf{b}_h + \boldsymbol{\delta}_{x,t}$
10: $d\mathbf{w}_h \leftarrow d\mathbf{w}_h + \boldsymbol{\delta}_{x,t}\mathbf{h}_{t-1}^\top$ ▷ add contribution to hidden state weights
11: $\boldsymbol{\delta}_{o+h,t-1} \leftarrow \mathbf{w}_h^\top \boldsymbol{\delta}_{x,t}$
   return $dL = [d\mathbf{w}_h\ d\mathbf{w}_x\ d\mathbf{w}_o\ d\mathbf{b}_o\ d\mathbf{b}_h]$

This implementation of back-propagation makes clear a problem that was explored by Hochreiter in 1991 [30] that occurs in training RNNs (and with very deep networks). The problem is that of gradient vanishing. This will be shown by examining how the error at q time steps prior to T is scaled by the weights $\mathbf{w}_h$ from propagating backwards through time. By looking at lines 5, 6, and 10 from Algorithm 3, the rate of change of the back-propagated

error $\boldsymbol{\delta}_{o+h}$ at time t due to the initial error at time T can be found as

$$\frac{\partial \boldsymbol{\delta}_{o+h,T-q}}{\partial \boldsymbol{\delta}_{o+h,T}} = \prod_{\mathrm{t}=T-q}^{T} \mathbf{w}_h^\top \psi_h'(\mathbf{z}_{x,\mathrm{t}}). \tag{2.21}$$

This is due to the fact that the back-propagation operations are all linear with respect to the error and $\frac{\partial \boldsymbol{\delta}_{o,t}}{\partial \boldsymbol{\delta}_{o+h,T}} = 0$ since it is not dependent on the back-propagated error from the previous time step. From this, it can be seen that if the largest eigenvalue of the the weights $\mathbf{w}_h$ is less than 1, the error will quickly vanish towards 0 resulting in extremely slow learning. If it is above 1, the error will blow up and cause very unstable conditions for gradient updates.

2.3 Long Short-Term Memory

Long Short-Term Memory (LSTM) cells were proposed by Hochreiter and Schmidhuber in 1997 to combat the gradient vanishing problem[31]. Since then, LSTM Networks have been shown to provide an immense performance boost over pure RNNs in speed of training and length of effective history. These new cells work similarly to RNN cells, but they introduce gated units that are used to ”decide” what information should be updated at each time step. These ”gates” are simply implementations of a FFNN layer that specifically uses the sigmoid element-wise activation function. This is due to its output between 0 and 1 that can determine if a value should be completely changed ($\psi(z) = 1$), left completely untouched ($\psi(z) = 0$), and everything in between. It makes use of these gates to determine to which level an added memory state should be modified that is shared between LSTM cells similarly to the hidden state in a RNN (LSTMs still maintain and utilize this hidden state). The difference with this new memory state is that it is only modified by very basic linear operations depending on the output from the gates.

For cleaner diagrams, the output layer that is included in the RNN cell in Figure 2.4 is moved outside and is not shown for the LSTM cell shown in Figure 2.5, but it is still used

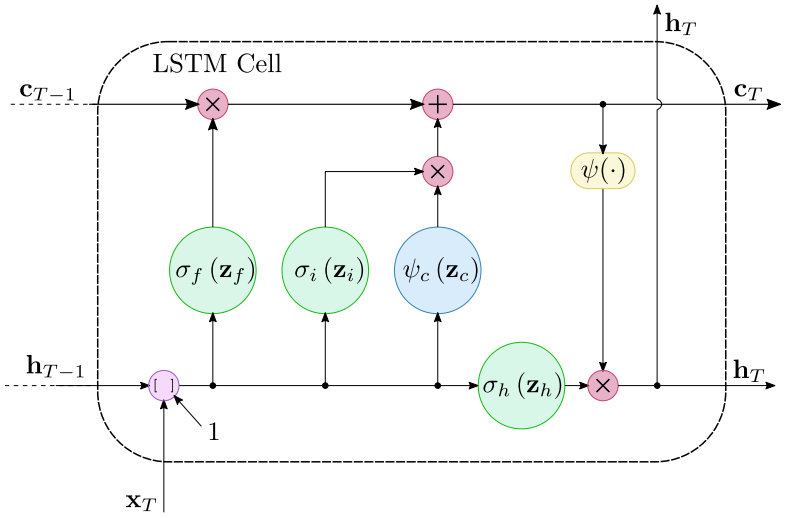


Figure 2.5: LSTM Cell.

in the same way as for RNN cells. For the LSTM diagram, the sigmoid $\sigma(\cdot)$ activations are the gates and the typical generic $\psi(\cdot)$ notation is dropped as these activations cannot use any other function to serve their purpose. The [ ] node in the lower left of the cell indicates the concatenation of all incoming signals. The incoming 1 to this node serves to simplify the math such that the individual activation layer biases are included in their corresponding weight matrices. This gives

$$\mathbf{z}_* = \mathbf{w}_*[\mathbf{h}_{T-1}; \mathbf{x}_T; 1] \tag{2.22}$$

for the input to each activation layer where $*$ indicates either the forget gate σ_f, input gate σ_i, output gate σ_h, or the new memory value ψ_c. Each red linear operation is an element-wise operation. This indicates that the number of units in each activation layer must be equal to ensure matching shapes at these nodes. The multiplication nodes can be seen as applying the outputs of the preceding gate as a filter to the other operand because they will only allow a certain percentage of each value through. It is in this way that they act as gates. First it can be seen that the outputs of the forget gate filter the memory state of the previous LSTM cell, therefore, "forgetting" some of the past. The input gate is then used to

filter the new memory states created by the ψ_c activation layer before they are added to the existing memory states. Finally, a copy of the memory state is run through an element-wise activation function (this is not a full neural layer, i.e. no weights), shown in yellow, that is typically chosen as $tanh$. This is filtered using the output gate and becomes the new hidden state $\mathbf{h}_T$.

The reason this new, much more complicated, cell structure is able to solve the vanishing gradient problem is that the error propagated backwards through time is able to maintain its magnitude through the memory states. This is what Hochreiter termed a constant error carousel (CEC). The error is only superimposed with new errors each time step from what is let through the gates. There is no resulting recursive multiplication of the error as was shown for the RNNs in (2.21).

2.4 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are another specific take on the FFNN that are different only in that they use a specific layer type that goes much further beyond the standard vector of parallel neurons introduced in Section 2.1 and utilizes the weight sharing technique introduced in RNNs. Often times when using neural networks for learning a task, it would be desirable to try to get the network to identify certain patterns in the input data to help it obtain the correct output. For example, when using pixels from a photo as input, it may make sense to find any vertical lines within the photo to help the network classify an object (maybe a simple box or package) in the photo. With the FFNN, it is possible given enough training data and a complex enough network to learn this classification task without explicitly searching for vertical lines or edges, however, it may take a very long time to converge to acceptable results. This can be avoided by taking a more direct approach to the problem at hand by directly scanning the input data for the desired pattern that is intuitively known to help the network converge. This is what the convolutional layer in a CNN attempts to do and when LeCun introduced this structure in 1989, he successfully

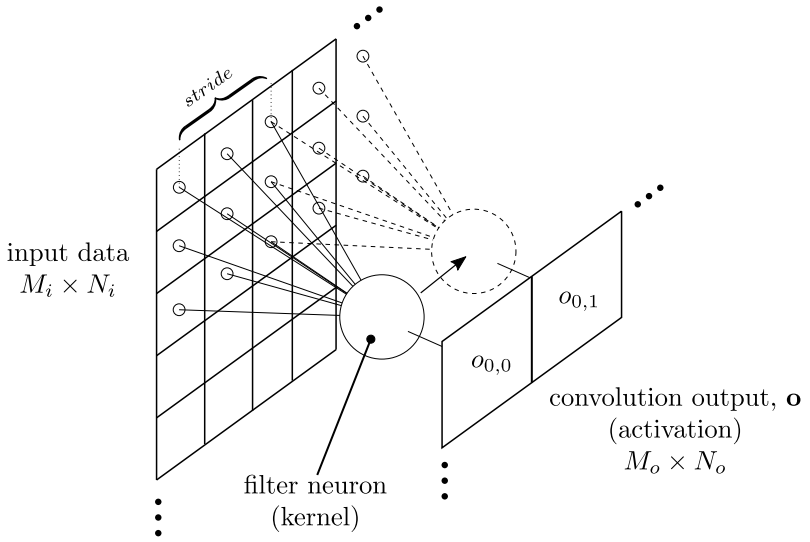


Figure 2.6: Filtering Neuron

used it to identify handwritten digits from images [32].

Effectively, a filter is convoluted with the input data to generate a new output of typically a smaller dimension that can be considered the activation of the filter. The reason for this is that the filter is actually a neuron with an activation function whose connections to the input data are such that they define the window of the filter. These filters are also known as *kernels* or *feature maps*. In Figure 2.6 a sample input of 2D data (or neurons) is shown and represented in a grid-like form with a filter using a window size of 3×3. The distance between each step of the filter is known as the *stride*. A horizontal stride of 2 is shown. Both the sizes of the filter and the stride control the output size of the convolution[1]. This is because the filter must always have a full window and cannot overrun the input data range, however, it is often desirable to generate a larger output shape and this is done by *padding* the input data (i.e. extending each dimension) with extra data, usually zeros.

In practice, the neuron is not actually moved across the input data, but instead there is a neuron for each output of the convolution and all neurons in that convolution share the same weights. This allows for static compilation and parallelization of the network, which

[1]This is not the same convolution operation as is used in statistics.

in turn leads to faster computations. Assuming a 2D convolution (as introduced above) with a kernel shape (window) $l_v \times l_h$ and depending on the stride lengths, a fixed activation output shape of $o_v \times o_h$ is given. The activation of the neuron in the m-th row of the n-th column of the output is given by a slightly more generalized representation than that provided in [33]:

$$o_{\mathrm{m,n}} = \psi \left(b + \sum_{\mathrm{j}=0}^{l_v-1} \sum_{\mathrm{k}=0}^{l_h-1} w_{\mathrm{m,n}} a_{(\mathrm{m}+s_v+\mathrm{j}),(\mathrm{n}+s_h+\mathrm{k})} \right) \tag{2.23}$$

Note that all indices here are zero-based. $a_{x,y}$ is the activation from the corresponding input neuron and s is the stride length where the subscript h or v gives either the horizontal or vertical direction, respectively. The bias for the kernel is given by b.

Conveniently, it is not necessary to explicitly specify the pattern (weights) that each filter uses. Since the output of the convolution layer is linear with respect to the individual weights, the back propagation algorithm introduced in Section 2.1 can be used to update the weights using gradient descent just as for the general FFNN. In this way, the network learns the proper filters to apply in order to identify features it deems most influential for the next layer.

In practice, it is often desirable to find multiple patterns in a single input set, so to do this multiple kernels are utilized and their outputs are stacked, thereby increasing the output dimension of the CNN layer. A rough visual assuming this condition with four kernels is shown in Figure 2.7. To bring the outputs of these stacked layers to the final desired output, a normal FNN layer is used connecting to all the outputs of the CNN layer. As can be imagined, it will not take much before the number of trainable weights needed to connect the new layer to the CNN grows to an enormous size. For this reason, a *pooling* layer is often added to simplify the outputs of the kernels.

The purpose of the pooling layer is to simplify the size of the CNN output. This typically means reducing it, but in some cases it could mean bringing a dynamic CNN output

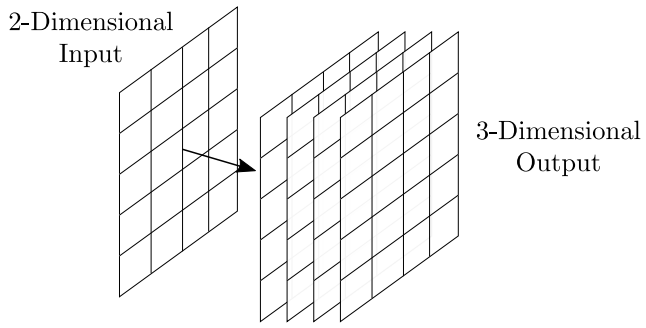


Figure 2.7: Dimensionality Change Using Multiple Kernels

shape to a consistent size [34]. In most cases a max-pool is used where the output kernels are sub-sampled and the maximum activation of each sample is used as the output. This is done commonly in pure classification tasks where the goal is just to identify if a particular feature exists in the data.

2.5 Temporal Convolutional Neural Network

Temporal Convolutional Networks (TCNs) are a specific type of CNNs. TCNs attempt to use the pattern searching capabilities of a CNN in a time series application. This is useful for sequence data such as time-based (temporal) signals and speech structure. These networks use a specific definition for the window and stride of the convolutions that result in output data that is determined only from those data points preceding its position in the sequence known as *causal convolutions*. As it is stated in [35], there is no leakage of future information to the past.

The current TCN structure used in recent research, is that proposed by Bai in [35]. Bai essentially ”modernized” the time-delay network used by Waibel in 1989 with current day convolutional network and computing methods[36]. Bai showed that the TCN was able to outperform the RNN stuctures in many benchmark tests. Further recent works have continued to show positive results from the use of temporal based CNNs over RNNs[37][38]. A brief introduction of the structure introduced by Bai is provided here as it is used in this

work. Most of what follows is taken from [35].

First, it is necessary to state that the TCN works on two basic principles, one of which was mentioned above: there can be no leakage of information from the future to the past (causality), and the network must produce an output of the same shape as the input. As mentioned, the leakage issue is taken care of by using causal convolutions. The second point is done by using a 1D fully-convolutional network (FCN) described in [39].

2.5.1 Dilated Convolutions

In order for the TCN to learn long sequences (i.e. effects of beginning values on end values) the current time step output must be in some way linked to the input values at the beginning of the history window. In the case of using CNNs, this can be done by stacking causal convolutional layers until the kernel size and stride (equal to 1 in TCNs) cause the current time step output of the last layer to link (through inter-layer connections) to the oldest time step input. Using this method, increasing the history so longer sequences may be learned will result in a linear increase in the number of stacked layers and the total network size (number of weights) will blow up. This can be avoided by utilizing *dilated* convolutions.

Dilating a convolution simply means skipping connections using the defined kernel size k. Here, a dilation $d = 2$ means that every second input is used in the kernel. Figure 2.8 shows how dilations can be used in stacked convolutional layers to efficiently extend the "hindsight" of the current time step output. Using a dilation factor of 2^{i}, for $\mathrm{i} = 0, \ldots, n-1$ with n stacked layers, as shown, is the standard method used to ensure that every input is considered in the output. In Figure 2.8, connections for the first (solid line) and second (dotted line) output activations are outlined. Note the use of one-sided zero padding to ensure the output is of the same shape as the input.

Due to the fact that TCNs can still become very deep networks, despite dilation, in order to capture a desired history length, they are still plagued by the problems previously

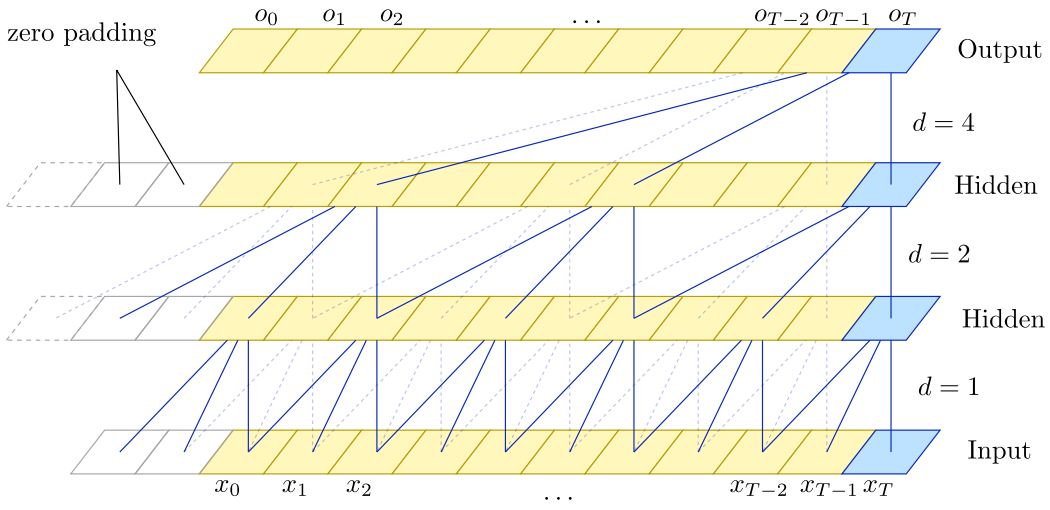


Figure 2.8: TCN mapping using only causal convolutions. Cf. with [35]

mentioned regarding RNNs. The fix Bai implemented to help with mitigating these issues was the use of residual blocks.

2.5.2 Residual Blocks

Residual blocks are introduced in [40] by He as a method to speed up the convergence of a network's learning by attempting to learn a difference between the input and output instead. In other words, rather than learning $\mathcal{F}(\mathbf{x})$, which directly maps the input $\mathbf{x}$ to the output $\mathbf{o}$ of a network, a difference mapping $\mathcal{H}(\mathbf{x}) := \mathcal{F}(\mathbf{x}) - \mathbf{x}$ is learned. This is based on the assumption that the difference mapping $\mathcal{H}(\mathbf{x})$ is close to zero, which should result in smaller weight values and, therefore, less iterations of gradient descent to push the weights to convergence. $\mathcal{F}(\mathbf{x})$ can now be redefined using the *residual* map

$$\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) + \mathbf{x}$$

where $\mathcal{H}(\mathbf{x})$ is the effective network structure to be learned. Even if this assumption is not true, the network will still be able to learn, it just may not be any quicker than training on the direct mapping.

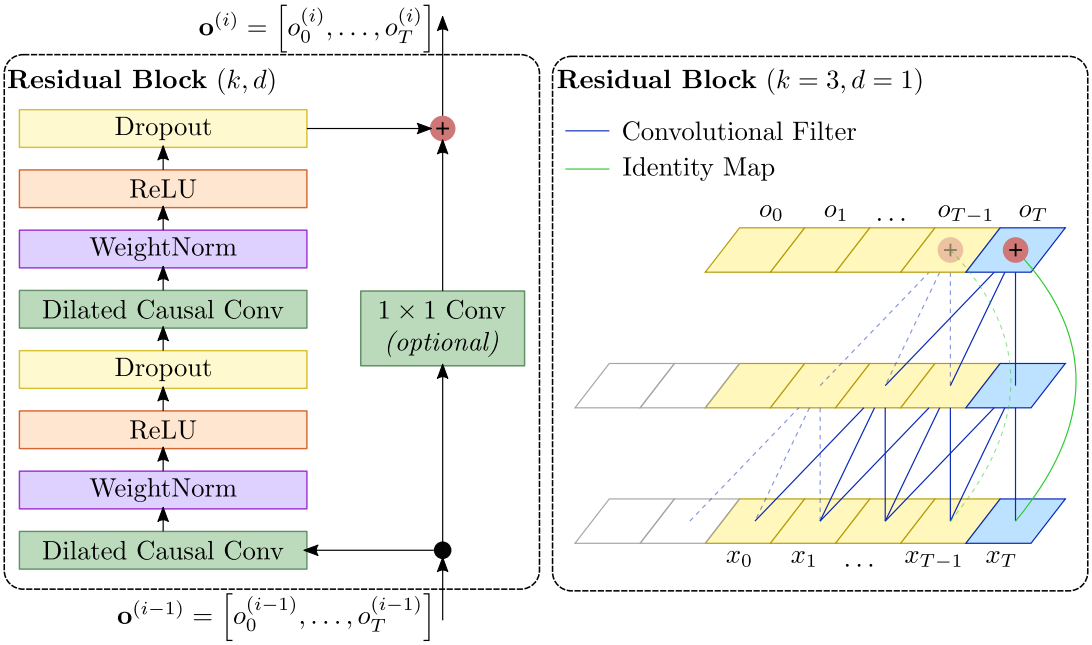


Figure 2.9: TCN generic residual block (left) and an example mapping for $k = 3$ and $d = 1$ (right). Cf. with [35]

As it is unlikely that the actual difference mapping is close to zero (i.e. the output is similar to the input) for deep, complicated networks, He breaks the full network into multiple, more shallow networks placed in series that utilize identity shortcuts to pass the input of each sub-network to its output. This can be seen as the green line on the right in Figure 2.9, which shows the residual block structure that is used by Bai for the TCN. These residual blocks are stacked in place of the single causal convolutional layers seen in Figure 2.8.

The 1×1 convolution on the identity map is noted as "optional" because it is only necessary to perform if the network structure on the left side of the residual block results in a different shape than the input. It is this structure (Figure 2.9, left) that holds the trainable network and other operations that are added to aid the TCN's convergence and results.

First, it is worth noting that there are two causal convolutions of the same dilation in each residual block. This makes the network more complex and gives it the ability to learn more complicated relationships for each dilation. "WeightNorm" is a weight normalization operation that is similar to the common batch normalization of inputs, but keeps the weights

independent of the inputs [41][42]. Both of these methods have been shown to speed up learning. "ReLU" refers to the rectified linear unit, which acts as the activation function for each convolution layer (see Table 2.1) [43]. Finally, "Dropout" refers to the common regularization technique of randomly disconnecting neurons during training[44].

2.6 Deep Adaptive Input Normalization

The last unique layer structure to be introduced, deep adaptive input normalization (DAIN), was developed in [45]. This layer attempts to solve a particular issue that arises when dealing with temporal input signals that are non-stationary. It has been long shown that convergence of a network is significantly faster and more likely when the inputs have been scaled to a similar magnitude (typically between -1 and 1) and/or normalized (i.e. center data about zero). This is particularly due to the fact that most activation functions have a steeper derivative value near zero, so the gradient updates will be more effective.

Normalization is a very simple operation when performing tasks such as image classification where the input signals (pixel values) have a fixed range, or doing predictions on a value that has natural statistical characteristics, such as temperature prediction during the summer. However, it doesn't make sense to apply the same to a signal such as product orders as analyzed in [45] or on signals such as the accelerator pedal input on a vehicle because it is extremely unlikely that the global statistics represent the current batch input being fed into the network. This means that there is a good chance of poor generalization when using the network on unseen data. The DAIN layer attempts to solve this problem.

The DAIN layer is placed as the input layer to a neural network and is used to learn scaling and shifting values dependent on the current batch data sample. It does this first by assuming that the data is generated as a Gaussian Mixture Model described as

$$p(\mathbf{x}) = \sum_{\mathrm{i}=1}^{N} \phi_{\mathrm{i}} \mathcal{N}(\boldsymbol{\mu}_{\mathrm{i}}, \Sigma_{\mathrm{i}}), \tag{2.24}$$

where ϕ_{i} is the weight of the i-th Gaussian with mean $\boldsymbol{\mu}_i$ and covariance Σ_{i} [45]. The normalizing of the signals is done using

$$\mathbf{x}' = (\mathbf{x} - \boldsymbol{\alpha}(\mathbf{x})) \odot \boldsymbol{\beta}(\mathbf{x}), \tag{2.25}$$

where $\boldsymbol{\alpha}(\mathbf{x})$ and $\boldsymbol{\beta}(\mathbf{x})$ use the current input $\mathbf{x}$ and trainable weights along with a *summary representation* of the current sample. These are given as

$$\boldsymbol{\alpha}(\mathbf{x}) = \mathbf{W}_{\alpha}\mathbf{s}_{\alpha} \in \mathbb{R}^d \tag{2.26}$$

$$\boldsymbol{\beta}(\mathbf{x}) = \sigma(\mathbf{W}_{\beta}\mathbf{s}_{\beta} + \mathbf{b}_{\beta}) \in \mathbb{R}^d. \tag{2.27}$$

Here, $\mathbf{W}$ is the trainable weights that can be trained using BP and the summary representations are given by

$$\mathbf{s}_{\alpha} = \frac{1}{L}\sum_{\mathrm{i}=1}^{L}\mathbf{x}_{\mathrm{i}} \in \mathbb{R}^d \tag{2.28}$$

$$\mathbf{s}_{\beta} = \frac{1}{L}\sum_{\mathrm{i}=1}^{L}(\mathbf{x}_{\mathrm{i}} - \boldsymbol{\alpha}(\mathbf{x})) \in \mathbb{R}^d. \tag{2.29}$$

Only the general method is provided here. For a more detailed description of DAIN, the reader is referred to the original publication [45].

CHAPTER 3

MOTOR CONTROLLER MODELING USING B-SPLINE PREDICTION

Series prediction problems tackled with the neural network approach are susceptible to certain difficulties not present when the same is applied to simple classification problems. These difficulties stem directly from two factors, the desired dimension of the output space and the output values' dependence on any neighboring values.

The downside to a large output dimension is simple: as the dimension of the output space increases, so does the number of weights needed for training to connect the output layer to its preceding layer at a larger than linear rate. This will lead to longer training times and more necessary computing resources for implementation of the network. This situation occurs when the forecast length increases to predict further into the future and the step size remains the same in order to capture the general profile of the desired prediction and simplify training data preparation. This problem is most often overcome in practice by selecting a single desired output step and disregarding what may happen before that value. Others have found that decent results may be obtained with a quicker training speed by simply implementing the previously mentioned solution multiple times for each desired step to simulate a continuous prediction[20].

The second issue is apparent when looking at the given problem within this work: vehicle speed prediction. Intuitively, it is known that the speed of a vehicle at a given point in time is inherently related to its preceding speed because it is part of a continuous dynamic system. However, a neural network has no intuitive knowledge of this relationship and each output activation has no idea what the others are doing. This often results in extremely non-smooth, or "noisy," outputs, which is not the expected result. This problem is often ignored in practice and the output is just smoothed once it is obtained.

This chapter will present a novel B-spline knot prediction method to overcome these

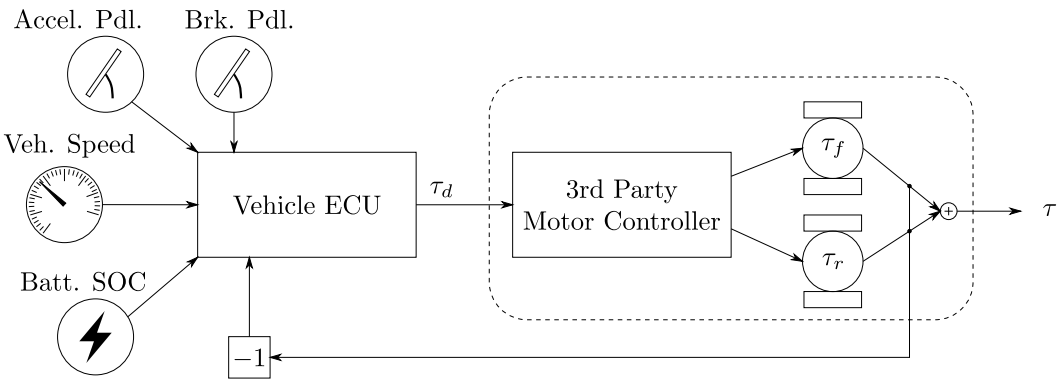


Figure 3.1: System configuration isolating 3rd-party controller.

shortcomings and apply it to the modeling of an unknown controller on an electric vehicle using a neural network. Using neural networks for such a modeling task is a natural choice as they are the ultimate black box tool when dealing with nonlinear control laws which these controllers are very likely to have. Given any vehicle, input signals to these controllers and the motor outputs may be measured, which provides all the necessary data for training a neural network using the methods explained in Chapter 2.

Initially it may seem superfluous for some parties, such as a vehicle's original equipment manufacturer (OEM), to need to implement such a method of modeling for their own product. Upon a closer look, however, it is seen that vehicles are often constructed from a multitude of third-party components each with their own built-in control systems and threshold limits that are proprietary knowledge, unknown by the manufacturer. This, for example, may be the case with the electric drive motors of an EV.

The presented network will be used to predict the true torque output from the motors of an EV given a requested input torque obtained from the mapping of the human-machine interfaces (HMI) (accelerator pedal, brake pedal, etc) made by the OEM onboard computer that is given to the 3rd-party controller and relevant motor states. An example diagram of the system can be seen in Figure 3.1. The neural network will be used to identify the section outlined with a dashed line.

First, a short explanation of the machine learning tool, TensorFlow, is provided to give the reader an understanding of how the presented methods are implemented.

3.1 TensorFlow

TensorFlow is a framework developed by Google LLC written mostly in C++ with a python interface. It's intention is to mix low level capabilities of designing systems and learning algorithms with high level predefined frameworks such as Keras, which is also used in this work.

TensorFlow operates by defining operations on a *graph* and connecting the tensors (inputs and outputs) of these operations in a flow-like structure. One advantage of doing this allows TensorFlow to be able to allocate some of these operations, namely matrix multiplication, automatically on a graphics card if present. This helps in particular to parallelize the back propagation during neural network training without any extra effort on the part of the user. More importantly, this graph-structure allows TensorFlow to utilize automatic differentiation. Automatic differentiation is a powerful computational tool in which the derivative of any operation may be found numerically[46]. This means that it is not necessary to manually define and calculate the derivatives of custom user-defined functions (although this is still possible to do), which is important when working with neural networks as derivatives are imperative for implementation of back propagation.

Keras, a built-in set of neural network layer structures and optimizing algorithms in TensorFlow, is used for the implementation of the network layers of the models described in later chapters. For additional capabilities not included in Keras, pure TensorFlow operations are defined and used. An example of this is seen directly in the next section and also for the implementation of the DAIN layer.

3.2 B-Spline Prediction Method

This work proposes a simple solution to both of the learning problems proposed at the start of this chapter through the use of B-splines. Under the assumption that the desired data to be predicted can be reasonably represented as a smooth continuous spline (i.e. not a

series of sudden jumps), which is typically the case when dealing with continuous-time dynamic systems, the proposition is to instead predict a minimum number of control points that can represent the data on the desired interval. In regards to the neural network output dimension size, this has the effect of increasing the step size of the output predictions so the output size can be smaller than the actual training data used for the prediction interval. Also, through utilization of the full B-spline equations in the loss function, each control point output becomes dependent on its surrounding training data points.

3.2.1 Cardinal B-Splines

B-splines, or *basis* splines, are piece-wise polynomials that are convoluted with each other to compose a single smooth resultant spline. The B-spline used in this work is a specific type known as a cardinal B-spline. Cardinal B-splines have the unique requirement that each knot (also acting as each control point in this case) is equally spaced along the desired interval. This allows for a fast implementation due to a cardinal spline's simplification by a simple shifting of the basis splines that then compose the complete spline rather than requiring new coefficients for each knot location[47].

If the sequence of knots is given by $\mathbf{t} = (t_{\mathrm{i}})_{\mathrm{i}=1}^{n+\mathrm{d}+1}$ where d is the degree order of the spline and n is the number of internal control points, then the B-splines composing a full curve may be computed recursively using

$$B_{\mathrm{j,d}}(x) = \frac{x - t_{\mathrm{j}}}{t_{\mathrm{j+d}} - t_{\mathrm{j}}} B_{\mathrm{j,d-1}}(x) + \frac{t_{\mathrm{j+1+d}} - x}{t_{\mathrm{j+1+d}} - t_{\mathrm{j+1}}} B_{\mathrm{j+1,d-1}}(x), \tag{3.1}$$

with

$$B_{\mathrm{j,0}}(x) = \begin{cases} 1, & if\ t_{\mathrm{j}} \leq x < t_{\mathrm{j+1}}; \\ 0, & otherwise. \end{cases} \tag{3.2}$$

Here, the denominator in each term of (3.1) may be replaced by h, because it remains a constant value for cardinal B-splines due to the fixed distance between knots[48]. These

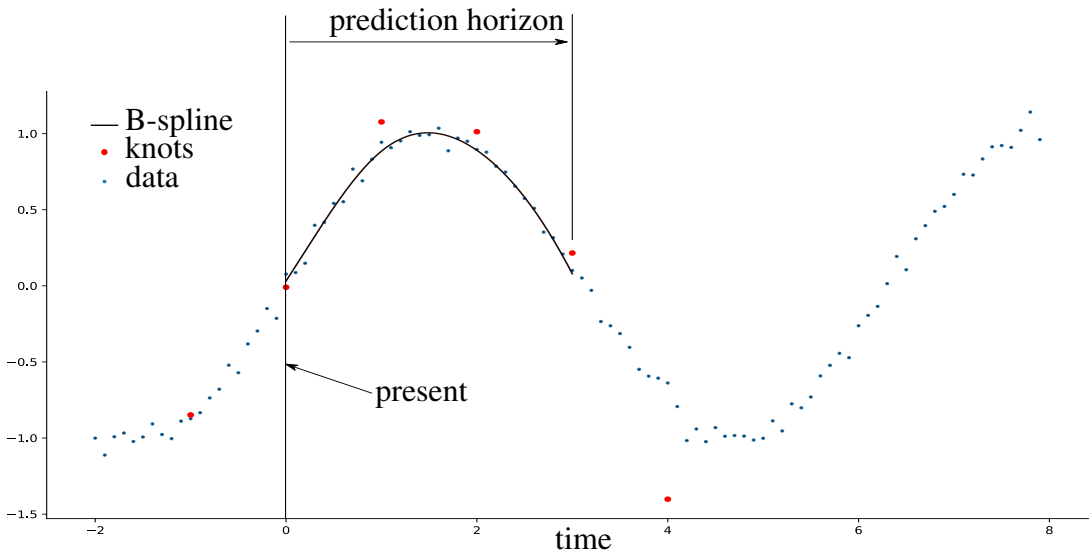


Figure 3.2: Example prediction using a B-spline.

B-splines are combined to generate the full curve S using

$$S(x) = \sum_{\mathrm{j}=1}^{n} \mathbf{c}_{\mathrm{j}} B_{\mathrm{j,d}}(x). \tag{3.3}$$

Here, $\mathbf{c}_{\mathrm{j}}$ is a coefficient matrix containing weights for the corresponding knots in $B_{\mathrm{j,d}}(x)$. These weights are what allow a B-spline to fit arbitrary curves. In this work, these weights are taken as the y-value in an x, y pair.

From (3.1) it can be seen that a B-spline requires knots on both sides of the desired location in order to calculate the spline value. This is seen in Figure 3.2 where a simple representation of this B-spline prediction method is shown. The sample shows how a B-spline can provide a parameterized, accurate representation of the actual signal with just 6 values. This new output now has the advantage of being continuous and differentiable without any additional post processing work.

3.2.2 Custom Loss Function

In order for a neural network in a supervised learning environment to learn these control points, first each training sample is fit with a B-spline and the training knots, $\mathbf{t}_t$, are stored. These "true" control points are used as the training values, however, the simple least-squares loss function introduced in Chapter 2 is no longer used directly on the output. Instead, test values along splines calculated from both the true control points and the outputted prediction points will be used for the square error calculation. These test values are given as

$$\mathbf{x} = [0,\ \Delta t,\ 2\Delta t,\ \ldots, T_p]\,, \tag{3.4}$$

where T_p is the prediction horizon of the network and Δt is a uniform step size given as $\Delta t = \frac{T_p}{N-1}$ for N test values. The knot vector $\mathbf{t}$ used in (3.1) is now taken as the NN output $\mathbf{o}$ or the training sample knots $\mathbf{t}_t$. Spline values are calculated for $\mathbf{y}_o$, the NN output, and $\mathbf{y}_t$, the training sample, using

$$\mathbf{y}_* = S(\mathbf{x}). \tag{3.5}$$

These spline values are then used with the typical squared error function defined in Chapter 2 to calculated the total loss of the prediction. This process can be seen in Figure 3.3. The derivative of this loss function is obtained through the use of TensorFlow's automatic differentiation capabilities so that it is a simple drop in loss function for the NN training tools provided in keras. To do this, a new B-spline fitting object was written using Tensorflow to fit control knots to the training data.

3.2.3 Notes on Performance

It is recognized that by using this method of output prediction, the neural network will be required to also learn the mapping of the B-spline equations. However, it is hypothesized that by using a sufficiently deep enough network by adding layers rather than just increasing their size, a proper balance between network complexity and output results can be found.

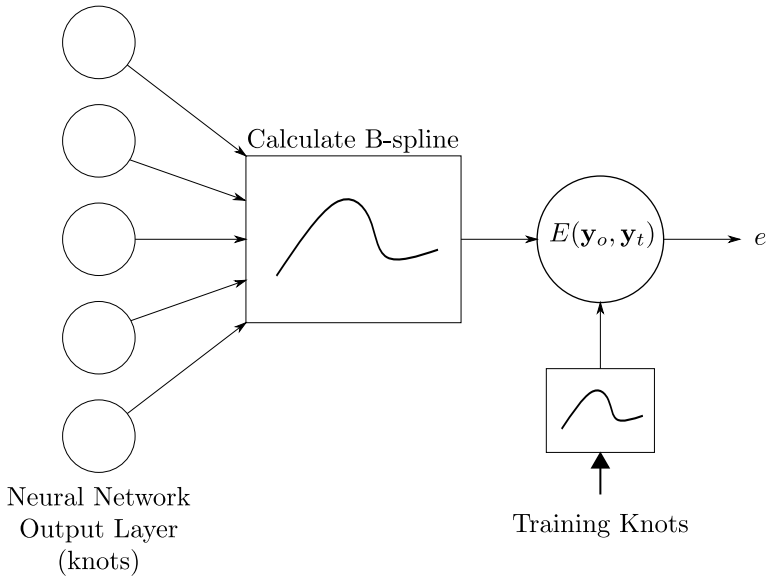
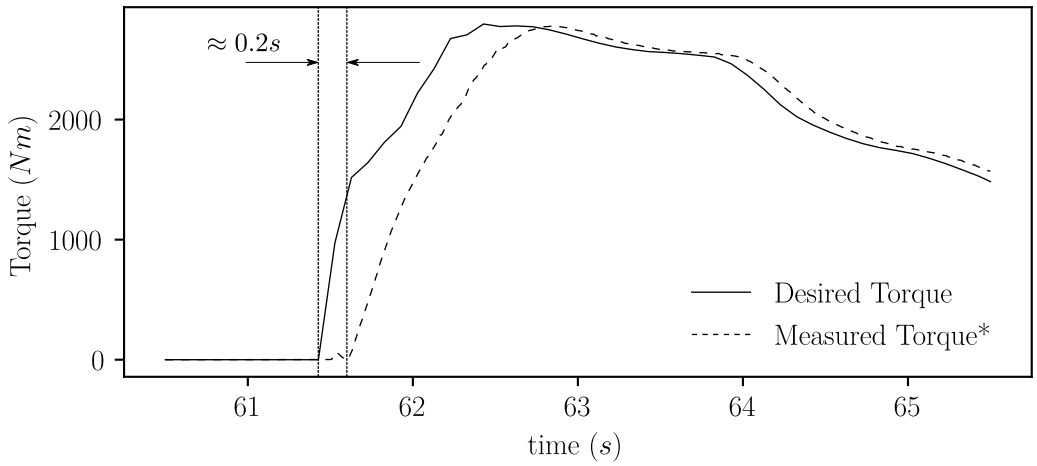


Figure 3.3: Custom loss function.

In an effort to reduce the training time, each B-spline training sample prediction is offset to start from zero by subtracting the present value. This forces the network to learn only how the signal will change as time progresses instead of learning the direct mapping from input values to output. This should reduce the variation of the magnitude of the outputted values. For example, when considering vehicle speed as the prediction signal, the values may be anywhere from 0 to 200 kilometers per hour. However, the immediate variation in speed is unlikely to have anywhere near as large of a range. Predicting the variation means the network weights will not need to have as extreme of values to do a direct mapping to such a large output range and therefore require fewer gradient steps before converging.

3.3 Motor Controller Network

For the motor controller identification task, an LSTM network is used due to its resemblance in implementation to that of a dynamic system. This is done because the network will be tasked in addition to identifying any controller laws with some basic system identification of the motors themselves. There is also a very apparent time delay between the given desired torque command, τ_d, and the measured torque from the motors that can be



* Values shown here are scaled from the true measured motor torque for visual purposes. Actual values are much lower as these measurements are taken directly at the motor before any gear ratios have been applied.

Figure 3.4: Time delay in application of commanded torque.

seen in Figure 3.4. This likely comes from computation times and inherent set update rates of the controller. An LSTM structure is also capable of learning this delay and this makes it the logical choice for capturing both of these features.

The structure of the network consists of using an LSTM cell that contains 256 units in the internal memory state followed by another 256 units in a fully connected layer. The final output layer of the network consists of 9 neurons representing the knots of the B-spline prediction. Between each layer a dropout connection is used that will drop 10% of the neuron connections during training to help prevent over fitting. The activation function used for the LSTM layer and the FC layer is the ReLU.

It is common practice when training neural networks to scale the input data such that it is near 0 and often shifted such that the mean of the data is zero. This procedure does not work well for the present task as this is non stationary time-series data. The mean values and other statistical characteristics from the training data will not accurately represent those of the validation data. For this reason, the DAIN layer introduced in Chapter 2.6 is used to scale the input signals and adaptively determine the scaling and shifting of the inputs during training. Inputs to the LSTM layer are summarized in Table 3.1 and a diagram of the full

Table 3.1: Input signals for motor controller identification.

Signal	Description	Units
Total Torque	Sum of the measured torque from both motors	Nm
Requested Torque	Requested torque value from the OEM computer	Nm
EV2 RPM	Rotational speed of the front axle motor	rpm
EV2 Voltage	Measured DC voltage at the front axle motor	V
EV RPM	Rotational speed of the rear axle motor	rpm
EV Voltage	Measured DC voltage at the rear axle motor	V
Battery Load	Current draw from vehicle battery	A

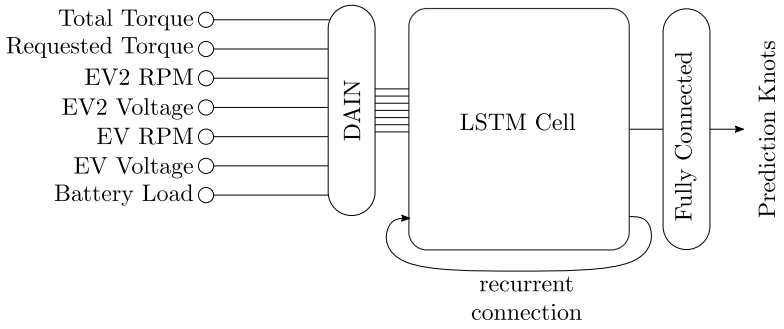


Figure 3.5: Network structure for motor controller identification.

network structure is shown in Figure 3.5. Additionally, the keras model output summary is included for reference in Appendix A.1.

A history of 0.5 seconds is used from each input signal in order to fully capture the previously mentioned delay in addition to any motor dynamics and all signals are discretized into 0.01 second time steps for input into the network. A 0.3 second output prediction horizon is used to try and identify the full dynamic interaction between an input command and when the effect is seen as realized in the measured output torque.

3.4 Training

A battery electric Mercedes-Benz EQC is used as the test vehicle and is equipped with two electric drive motors corresponding to the input signals as mentioned in Table 3.1: one for the front axle, one for the rear axle. Both motors are connected to their respective axles

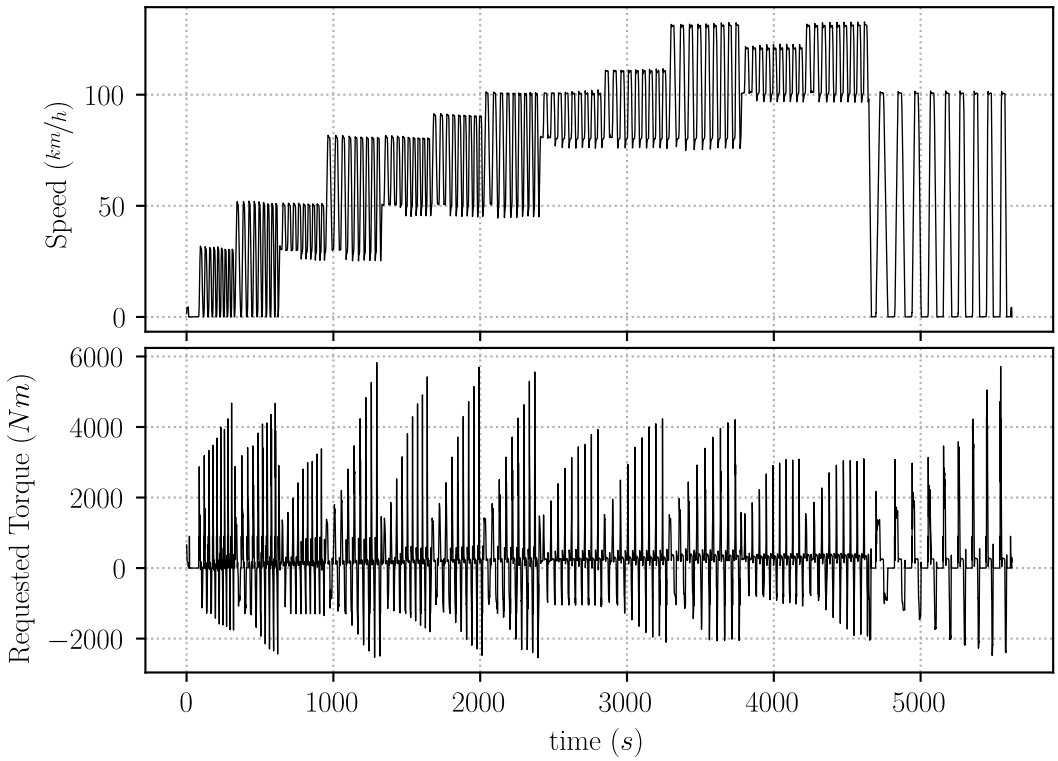


Figure 3.6: Torque and speed sweeps for training data.

using a differential gearbox. All vehicle signals indicated in Table 3.1 are available via a Controller Area Network (CAN bus).

The training data is obtained from vehicle testing performed on an electric dynamometer with the capabilities of simulating driving conditions such as road and wind resistance. Two complete sets of test data are gathered. The first, used as training data for the neural network, is a comprehensive sweep of torque commands between different set vehicle speeds. The intention is to obtain training samples for every condition that the vehicle will encounter under normal operation. The speeds and torques from the collected training data are shown in Figure 3.6.

The second set of test data is obtained through the simulation of a previously driven route. This data is used as the validation set during the training of the network and the speed and requested torque can be seen in Figure 3.7. From comparison with Figure 3.6 it can been seen that the typical driving condition set is fully enveloped by the torque sweeps.

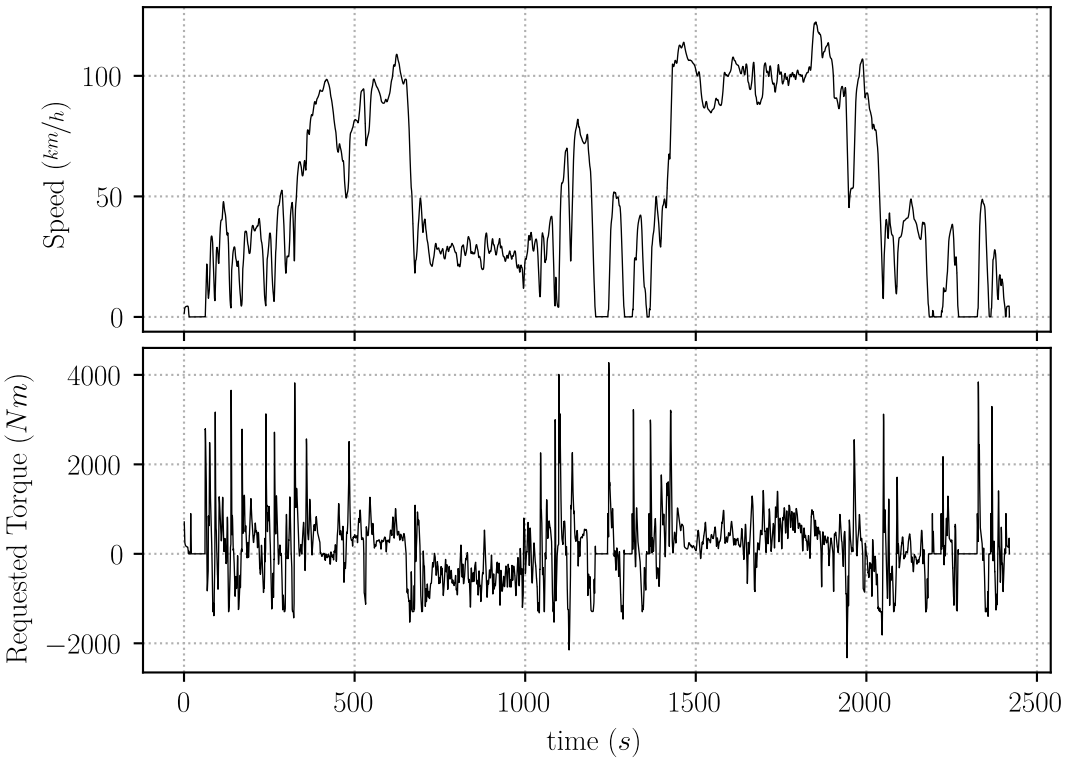


Figure 3.7: Torque and speed sweeps for validation data.

The optimizer tool used during training was the built-in keras optimizer, Adam[49]. The network is then trained for 34 epochs, or 34 times through the entire training set, with a batch size of 8 samples and the results of the mean value of the losses for both the training and validation set for each epoch is shown in Figure 3.8. The jump in learning at epoch 23 was caused by the need to restart training. This caused a jump because the Adam optimizer is stateful as it performs a type of momentum-based gradient calculation and the restart caused it to lose its current state. Due to the time cost of training, the results were deemed acceptable for the purposes of this investigation.

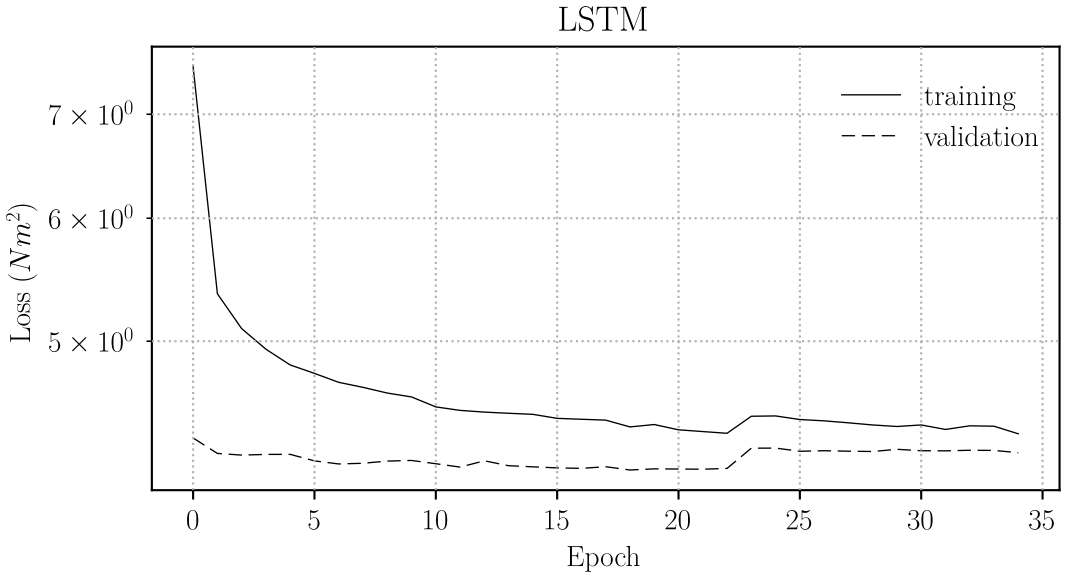


Figure 3.8: Training and validation losses for motor controller identification.

3.5 Results

For analyzing the results, the error at two points along the predicted B-splines are calculated: 0.1s and 0.3s. The error at these points is calculated simply as the absolute error, given as

$$\text{error} = |\tau_{\text{prediction}} - \tau_{\text{measured}}| \tag{3.6}$$

where τ_{measured} is interpolated linearly should the prediction point occur between data points.

The predictions taken every 0.2 seconds along the simulated route are shown in Figure 3.9 against the measured torque. The absolute errors are also plotted for the total predictions made every 0.01 seconds. At the 0.1 second prediction interval there are recurring spikes in the error of approximately 10-15 Nm. To help isolate what is causing these spikes, a small section of the route is isolated and shown in Figure 3.10. Also, there appear to be very large jumps in error at the 0.3 second prediction interval. One of these jumps is isolated and shown in Figure 3.11b.

In Figure 3.10 it can be seen that the network is able to accurately predict the total

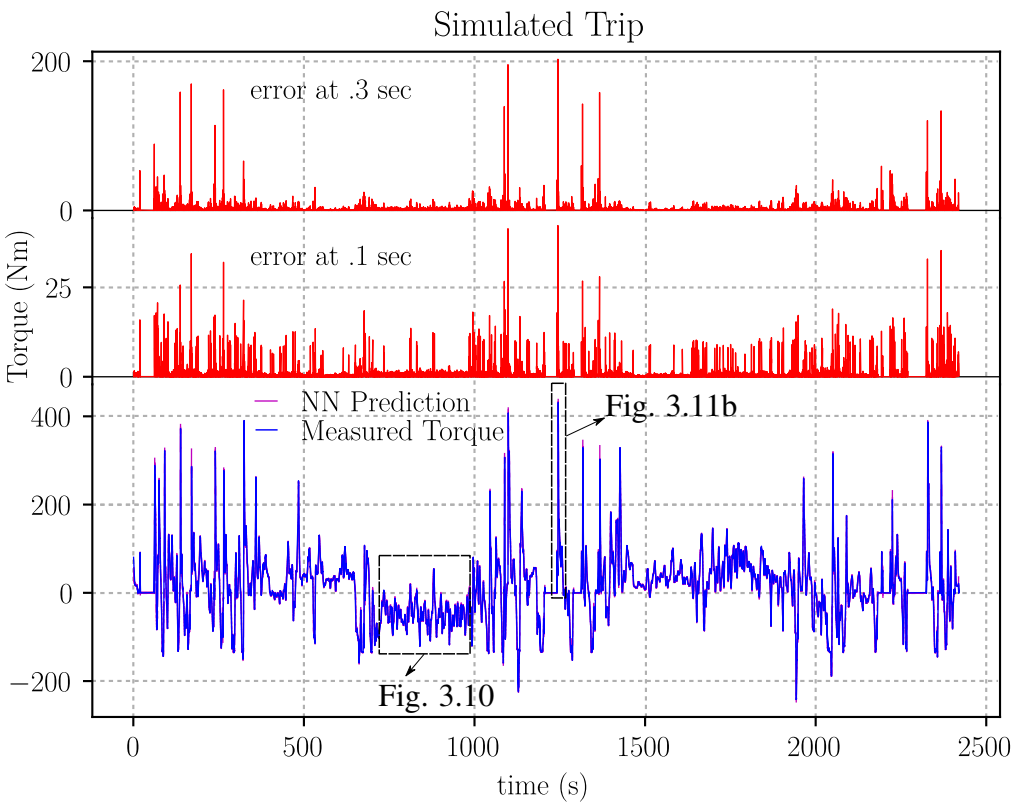


Figure 3.9: Network predictions and errors for simulated trip.

torque output of the motors in most cases with low error. However, there are spikes in the error at points where the torque output reverses direction from positive to negative and vice versa. At these points, the torque output has a tendency to jump one direction and then back. This can be seen more clearly in Figure 3.11a. This jump is easily explained by any backlash in the drivetrain that causes a dead zone of no load on the engine and then a sudden jerk of full load. The neural network failed to account for this phenomenon at all.

The cause of the very large jumps in error seen at the 0.3 second interval is very apparent when looking at Figure 3.11b. At drastic changes in the torque value, the network is unable to capture the near step-like signal value leading to huge offsets in the predicted torque versus the true value.

These results overall show that the presented method was able to learn the behavior of the system under most circumstances except occurrences of steep changes in the slope of

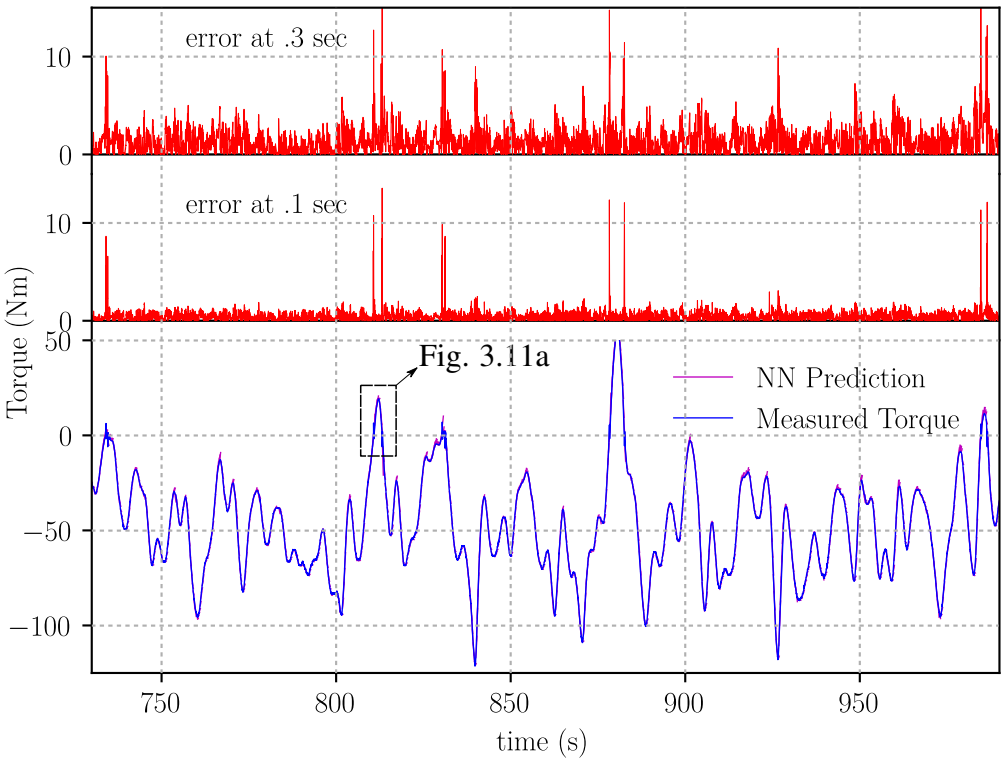
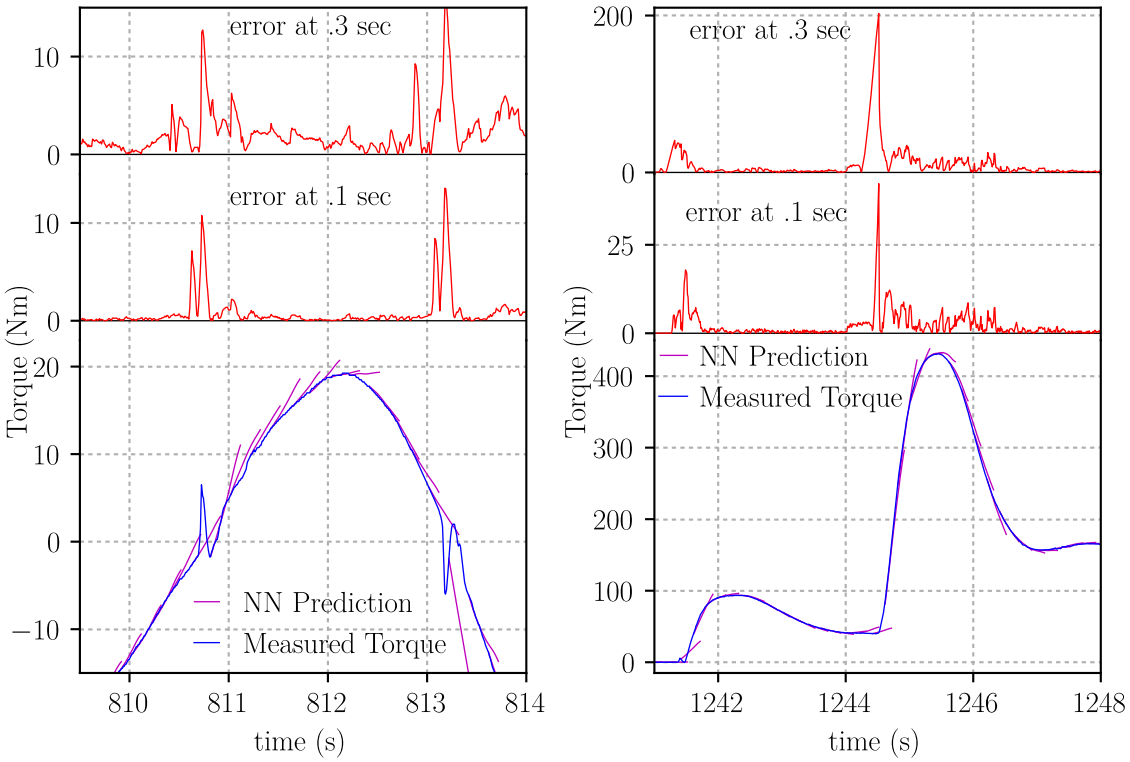


Figure 3.10: Isolated section of simulated trip with low error.

the output torque. This is likely caused by a shortcoming of the B-spline process. The number of controlling knots that the network was given to use was too low and did not allow for capturing these features in the data. In a sense, they were filtered out.

Due to the significant amount of time required for generating training data, further testing on this effort was not completed. This large time drain comes from the necessary use of the custom B-spline fitting algorithm that was written for this work. The algorithm is far from optimized and the process could be expedited using a commercial package for determining the training knots, however, this was not done due to the requirement that the algorithm needed to be implemented natively in TensorFlow so that the numerical derivatives could be tracked.

Overall, this prediction method is shown to be accurate for conditions of smooth operation with no large jumps in output. Unfortunately, this is not ideal for accurate modeling



(a) Error at zero crossing.

(b) Isolated large error spike.

Figure 3.11: Occurrences of high error.

of a system with higher frequency responses such as an electric motor as shown. A more encompassing approach to capture the presented issues would be a classical parametric modeling approach. However, the characteristics of the B-spline output are ideal for the task taken in the next chapter where high frequency responses, such as those shown here, are not as prevalent in the outputted signal and a parametric modeling approach is not very feasible for the presented prediction horizon.

CHAPTER 4

SHORT HORIZON SPEED TRAJECTORY PREDICTION

This chapter will focus on the task of vehicle speed prediction. A unique formulation of input data and neural network structure is investigated in order to capture both temporal and spatial affects seen by the vehicle at any instant in time. An example sketch of the given problem is provided in Figure 4.1.

Here it is possible to see how the driver will need to change their inputs to account for the upcoming stop sign. In the shown situation, that would include reducing the throttle and applying the brake as described by the ”Accel. Pdl” and ”Brk. Pdl.” signals, respectively, in the figure. There are numerous other factors that will affect the upcoming speed profile change that are not shown in Figure 4.1. These include factors such as driver reaction speed, current vehicle speed, the presence of any preceding vehicles, and distance from the sign. However, driver reaction times due to conditions like inattentiveness [50], age and experience [51], and aggressiveness [52] are not covered within the scope of this work. Measurable data from the vehicle sensors and known route information such as the other previously mentioned factors will be taken into account.

While there is no explicit driver characteristic inputs considered, the proposed approach will be expected to infer how a general driver responds to presented route conditions through extensive training data. This is because the prediction horizon used in this task is long enough to require consideration of the effects of the driver’s inputs.

The integration of traffic and geographical information with vehicle data to achieve accurate vehicle speed predictions will be done using a unique parallel input neural network. The proposed method will be evaluated against existing simple standard vehicle prediction methods. Also, a study of the direct effect of including route information as part of the network inputs is done through an isolation of the temporal signals as inputs.

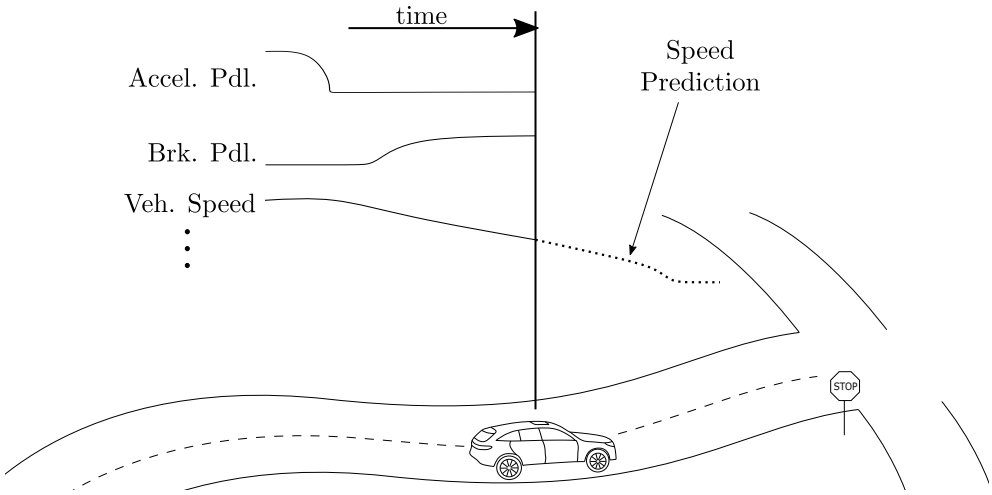


Figure 4.1: Vehicle state on street.

4.1 Route Analyzer Tool

In order to perform the inclusion of route data as mentioned, it is first necessary to obtain many details about a vehicle's driven route. These details include route features such as traffic signals, speed limits, etc. and are made available from HERE using a representational state transfer (REST) application programming interface (API) using standard internet request protocols (GET, POST, etc).

A graphical user interface (GUI) application, Route Analyzer, was developed concurrently with this work in order to access this data and combine it with the thousands of stored test vehicle trips available on a proprietary company server and can be seen in Figure 4.2. Route Analyzer was written fully in python using the pyqt framework. This allowed for simple native use of multi-threading operations to parallelize functionality while integrating everything with the GUI.

Route Analyzer uses the global positioning system (GPS) points from a route matching request sent to the HERE server and matches them in time to the recorded onboard GPS signals from a vehicle. Once this is complete, it is trivial to provide a time base to the other HERE data and combine all signals, from HERE and the vehicle, together in the same format for viewing and further processing.

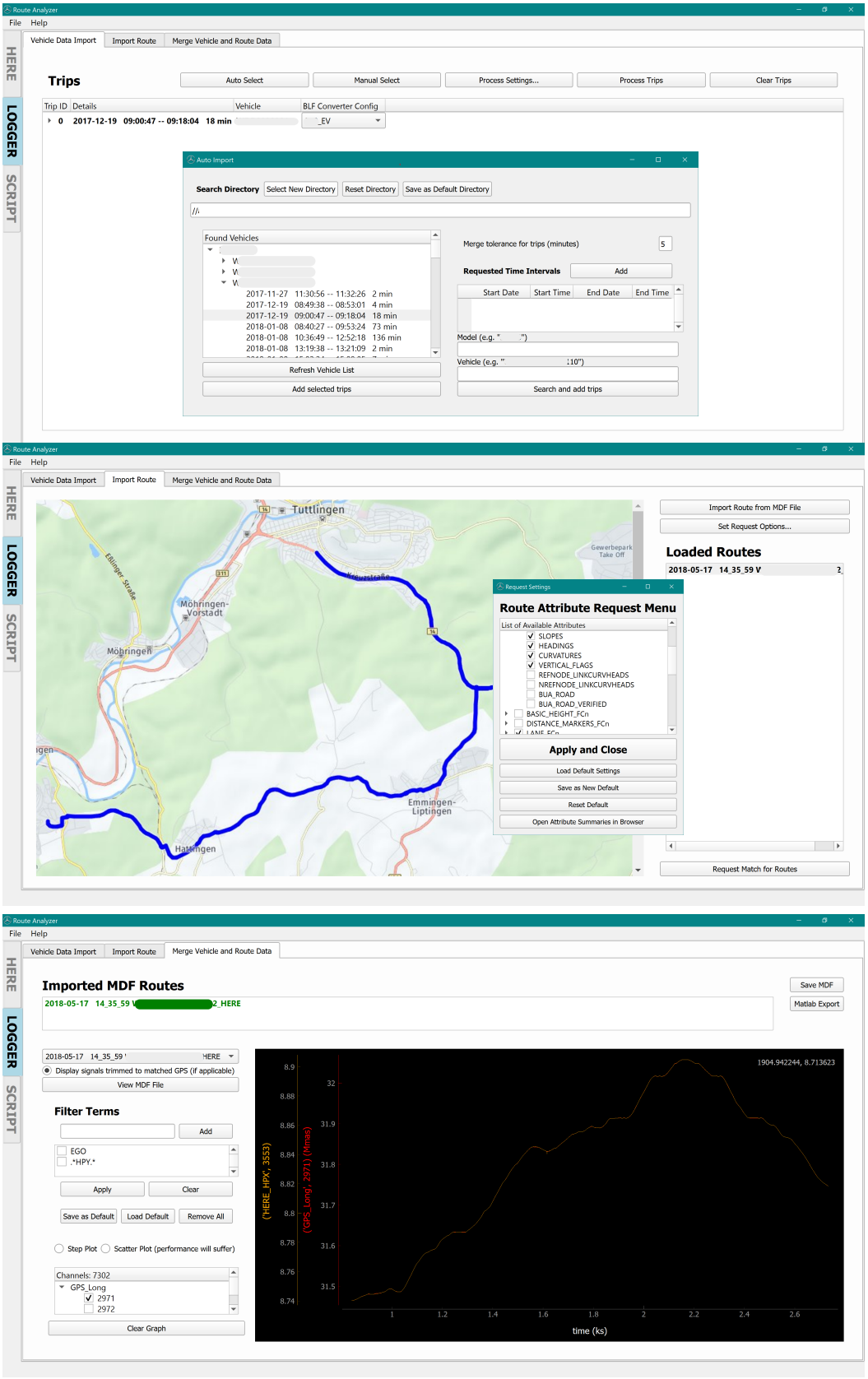


Figure 4.2: Route Analyzer GUI application.

4.2 Time Series and Distance Series Discretization

First, it is necessary to establish what data is considered pertinent to the problem and the corresponding proper input form to use with the neural network. The input data is first split into two groups, either temporal or spatial data, and from there it is discretized accordingly. Temporal signals are the signals relating to the inputs and states of the vehicle and are available onboard through the CAN bus. Spatial data comes from the surrounding environment such as speed limits, signs, etc. These signals are made available from the HERE REST API. This data is discretized using a distance measurement because it is impossible to know how these features relate to the vehicle in a time sense without already knowing the entire vehicle speed profile as it moves along the route. The work in [12] does convert the road information to a time base, but does so under the assumption of a known speed profile since it deals only with an ACC. Table 4.1 shows an overview of the input signals that are used along with brief descriptions. The scaling factor listed in Table 4.1 is used to bring the raw data signals to a similar magnitude and result in values between -1 and 1 using

$$\hat{s} = \frac{s}{\text{scaling factor}}, \tag{4.1}$$

where s is a sample data point and $\hat{s}$ is the scaled output. This will help with convergence as no signal is unintentionally weighted heavier than others at the start and therefore masking the effects of the signals of smaller magnitude.

4.2.1 Time Series

The temporal data input is dictated by the desired historical window size and the time step size to use for discretization. Given a time step Δt and k_d time steps into the past, the temporal data input for a single signal is given as

$$\mathbf{x}_{time,*} = \begin{bmatrix} x_{*,T-k_t\Delta t} & x_{*,T-(k_t-1)\Delta t} & x_{*,T-(k_t-2)\Delta t} & \dots & x_{*,T-\Delta t} & x_{*,T} \end{bmatrix} \in \mathbb{R}^{k_t+1}, \tag{4.2}$$

Table 4.1: Input signals.

Signal	Description	Units	Scaling Factor
Temporal Signals			
Vehicle Speed	Longitudinal velocity of the vehicle	$^{km}/_{h}$	200
Prec. Veh. 1 Distance	Distance to the preceding vehicle	m	204*
Prec. Veh. 2 Distance	Distance to the pre-preceding vehicle	m	204*
Steering Wheel Angle	Angle of the steering wheel in the cockpit	$^{\circ}$	360
Accelerator Pedal	Percentage throttle input given by driver	-	100
Brake Torque	Applied braking torque by the motors	kNm	4200
Brake Pedal	Digital value whether or not the brake pedal is being pressed	-	-
Spatial Signals			
Traffic Signals	Existence of stop lights at intersections	-	-
Stop Signs	Existence of stop signs	-	-
Crosswalks	Existence of crosswalks	-	-
Yield Signs	Existence of yield signs	-	-
Roundabouts	Existence of roundabouts	-	-
Speed Limits	Posted traffic speed limits	$^{km}/_{h}$	200
Average Speed	Historical average traffic speed	$^{km}/_{h}$	200
Road Curvature	Curvature of the road	$^{1}/_{m}$	-
Road Slope	Road grade in direction of travel	$^{\circ}$	20

* Approximate static measured value when no leading vehicle present.

and the complete temporal input is

$$\mathbf{x}_{time} = \begin{bmatrix} \mathbf{x}_{time,1} & \mathbf{x}_{time,2} & \dots & \mathbf{x}_{time,N_t} \end{bmatrix}^{\top} \in \mathbb{R}^{N_t \times k_t + 1}, \tag{4.3}$$

where N_t is the number of temporal input signals.

A time step $\Delta t = 0.01$ seconds is chosen as this is the finest sampling rate used in the recorded CAN bus data for training. Signals sampled at a slower rate are simply linearly interpolated when discretizing all signals as a whole. The window size for past data to include is chosen as 1 second ($k = 100$) as a reasonable length of time to encompass both human reaction speeds [53] and dynamics of the systems involved (speed changes of the EGO vehicle and preceding vehicles).

4.2.2 Distance Series

The spatial route data is organized in a similar format as the temporal data, however, the current time T is taken as the current position D. Another major difference is that the discretization step size is not constant. Instead, a dilating step size is used that is shorter near the current position and increases in length as it moves further away. Using a window that includes both values preceding the vehicle position and following the vehicle position, the relative discretization is chosen as a strictly monotonically increasing series and given as

$$\bar{\mathbf{d}} = \left[\begin{array}{ccccccccc} \bar{d}_{\mathrm{p}} & \bar{d}_{\mathrm{p}-1} & \dots & \bar{d}_{-1} & 0 & \bar{d}_{1} & \dots & \bar{d}_{\mathrm{f}-1} & \bar{d}_{\mathrm{f}} \end{array} \right], \tag{4.4}$$

where p is the number of preceding positions, f is the number of following positions, and 0 corresponds to the vehicle's current position. $k_d = \mathrm{p} + \mathrm{f} + 1$ is the total number of positions used ($\bar{\mathbf{d}} \in \mathcal{R}^{k_d}$). The absolute position discretization is given by

$$\mathbf{d} = \bar{\mathbf{d}} + D. \tag{4.5}$$

The relative distance discretization used in this work is

$$\bar{\mathbf{d}} = [\, -20 \underbrace{\dots -8 \; -6}_{\triangle=2m} \underbrace{\dots -1 \; 0 \; 1 \; \dots}_{\triangle=1m} \underbrace{6 \; 8 \dots}_{\triangle=2m} \underbrace{30 \; 35 \dots}_{\triangle=5m} \; 50 \; 75 \; \underbrace{100 \; 150 \dots}_{\triangle=50m} \; 400 \; 500 \,]m \tag{4.6}$$

with $\bar{\mathbf{d}} \in \mathcal{R}^{45}$, which gives a total history of 20 meters and look-ahead of 500 meters. Route information that occurs between positions that is binary in nature, such as the existence of a sign or traffic light, is assigned to the nearest existing position in the discretization. This is not expected to influence the results to a noticeable extent as the step size is chosen such that the difference in true position is smaller as it becomes more influential (i.e. as the vehicle is closer). The dilation is chosen in this way to reduce the number of inputs to the neural network and simultaneously obtain an effective window size.

The resulting spatial signal inputs are given as

$$\mathbf{x}_{dist,*} = \begin{bmatrix} x_{*,d_1} & x_{*,d_2} & \dots & x_{*,d_m} \end{bmatrix}, \tag{4.7}$$

where d_k is the k-th index in $\mathbf{d}$. The total spatial input to the neural network is

$$\mathbf{x}_{dist} = \begin{bmatrix} \mathbf{x}_{dist,1} & \mathbf{x}_{dist,2} & \dots & \mathbf{x}_{dist,\mathrm{j}} \end{bmatrix}^\top \in \mathbb{R}^{N_d \times m}. \tag{4.8}$$

Here, N_d is the number of spatial input signals to use.

4.3 Neural Network Structure

4.3.1 Full Network

It is not logical to process both input data groups the same way since they are built from fundamentally different bases. For this reason, a special temporal spatial neural network (TSNN) structure is proposed consisting of two parallel input network layers whose outputs are then concatenated together to further FFNN layers that will learn the relations between the data groups and determine the final output. This neural network is shown in Figure 4.3.

The temporal inputs are first fed into a TCN in order to extract relationships from the time series data. A TCN layer is used instead of a RNN structure due to the large data input size. The parallelization of the TCN with the standard BP algorithm is expected to speed up training over using a structure that requires BPTT. The TCN uses a kernel size of 8 with dilations of 1, 2, 4, and 8. This provides a history length of 64 time steps. This is larger than the selected data input history and will be compensated with zero padding as mentioned in Chapter 2. A total of 16 parallel TCN layers are stacked, each finding its own relationships in the data and ultimately returning 16 outputs.

The spatial data enters a simple CNN layer that uses a kernel size of 4 for each indi-

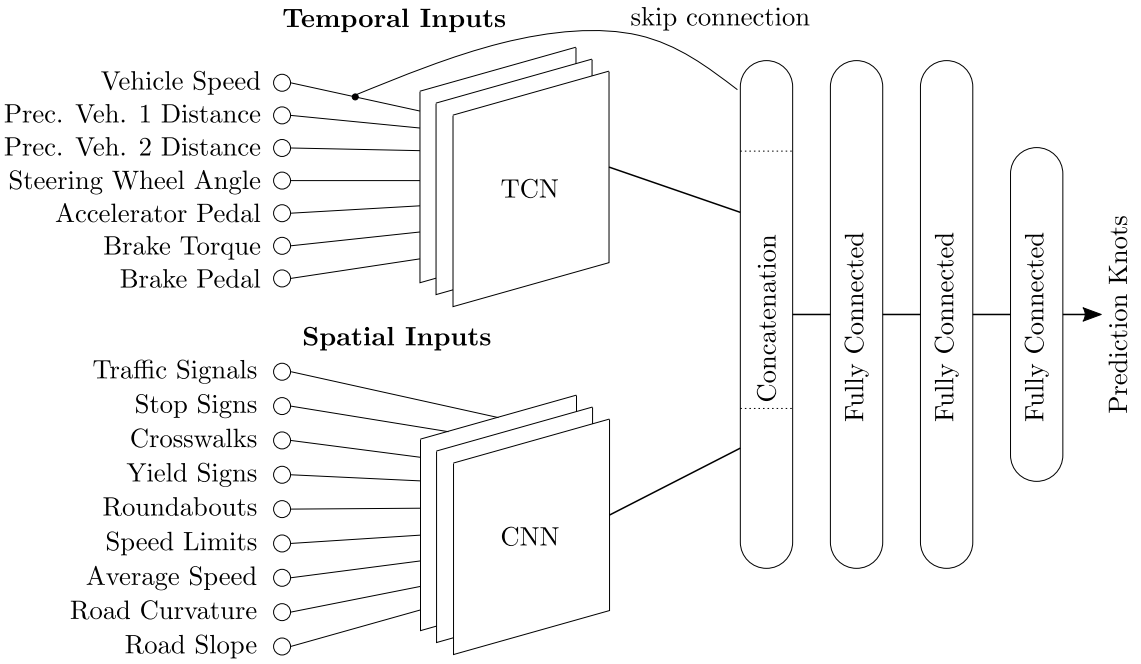


Figure 4.3: Full network structure.

vidual signal and a stride of 1 along the distance dimension. In other words, each signal receives its own kernel and there is no overlap of connections between signals. It is done this way to avoid erroneous activations due to similar patterns in the data but occurring across different signals. It is still desirable to use a CNN to take advantage of its pattern searching capabilities. Ideally, this portion of the network will recognize repetitious traffic conditions. 32 stacked CNN layers are used.

The outputs of the stacked TCN and CNN layers are flattened and concatenated before being connected to a simple FFNN with 3 layers. The first two layers have the same number of units at 256 and the third layer is the output layer with only 10, each representing a knot in the B-spline for speed prediction. In between all layers is a dropout connection set to randomly drop 30% of neuron connections during training to aid in preventing over fitting the data. In addition, the sizes of all layers mentioned above are chosen such that the validation data during training does not immediately diverge. For simplicity, the same activation function, the rectified linear unit, is applied to all layers.

It should be noted that a skip connection was included to pass a copy of the vehicle

speed input signal directly to the concatenation operation. This can also be seen in Figure 4.3 and was done to attempt to help the network learn a smoother continuity between the predicted speed profile and the existing past profile.

A summary of the complete model implemented in Keras can be found in Appendix A, where the distribution of trainable weights can also be seen.

4.3.2 Temporal Only

For comparison purposes, a network that utilizes only the temporal input from the full network structure described previously is implemented as well. This temporal-only neural network (TONN) is identical except the concatenation of the spatial data is omitted. This structure can be seen in Figure 4.4 and the Keras output summary is provided in Appendix A.

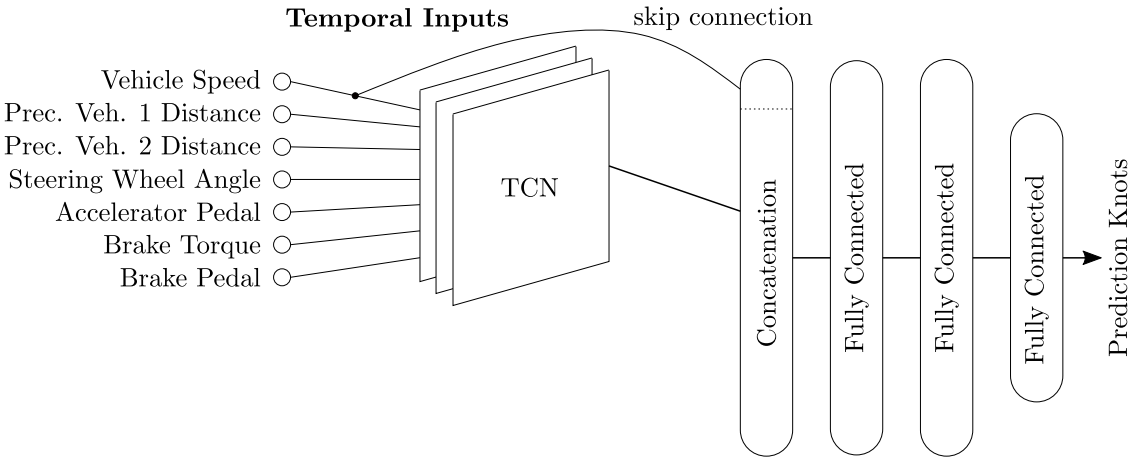


Figure 4.4: Temporal-only network structure.

4.4 Data Preparation and Training

The available vehicle test data for training the network does not utilize the same driver for all trips. Due to this, it is expected that there will be more variation in speed change rates given identical situations depending on the aggressiveness of the driver. Following the work done by Morlock and Rolle in [22], a compromise is made to attempt to alleviate the effects of these variations. A division of traffic conditions by road class is introduced using a classification metric provided by HERE that is outlined as:

- Class 1: a road with high volume, maximum speed traffic
- Class 2: a road with high volume, high speed traffic
- Class 3: a road with high volume traffic
- Class 4: a road with high volume traffic at moderate speeds between neighborhoods
- Class 5: a road whose volume and traffic flow are below the level of any other functional class.

Identical networks following the description previously outlined, are trained on each road class for a total of 5 different neural networks.

4.4.1 Preparation

For training the neural network, portions of two different available routes are used that were driven by the same vehicle. Figure 4.5 shows the different routes that were driven and the breakdown of the data between training and validation is shown in Table 4.2. The Route B trips on August 21 are trimmed to stop at Leonberg due to the presence of a tunnel increasing GPS match difficulties. The remaining portion of Route B beyond Leonberg consists of mostly highway driving, which is already sufficiently represented in the gathered data.

Trips that are marked for ”training” are used during the training of the neural network and back propagation is performed using the calculated loss from these trips. ”Validation”

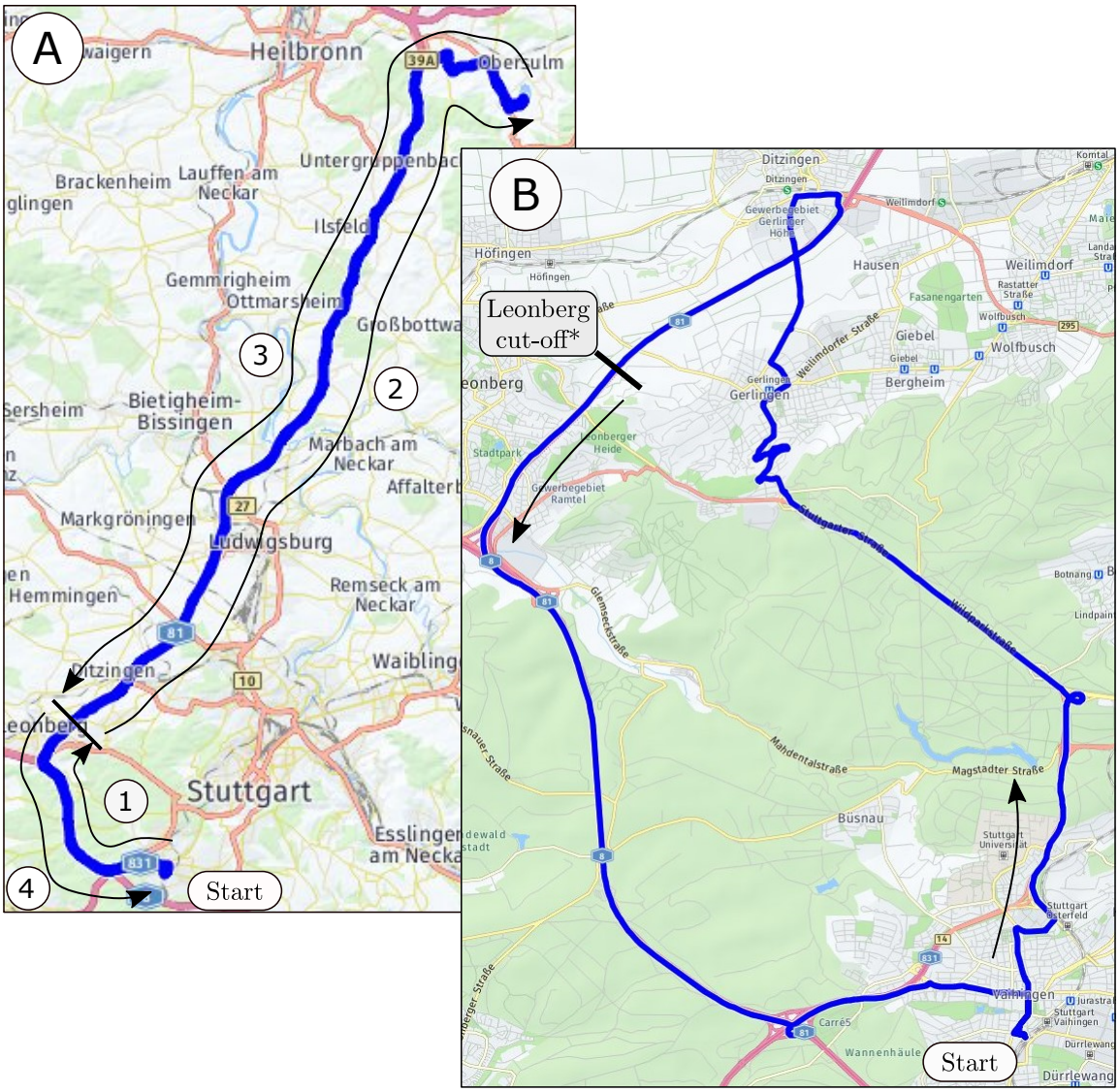


Figure 4.5: Routes used for evaluation. A) Vaihingen to/from Breitenauer See. B) Round trip from Vaihingen to Ditzingen.

trips are also used during training, however, the loss is calculated and only recorded for later viewing and back propagation is not performed using this data.

The data preparation for each trip first begins with identifying a valid test trip from the proprietary company database. This involves ensuring that the vehicle in question traveled somewhere relevant for training. For example, data with extensive driving in a parking lot, or on a test track is disregarded. Since a significant amount of the vehicle data originates from prototypes, the data must be checked to ensure all required signals for training are

Table 4.2: Driven routes.

Date	Start Time (UTC)	Route	Description	Use
September 5, 2018	12:45	1A	route to Breitenauer See, start section	Training
September 5, 2018	-	2A	route to Breitenauer See, end section	Validation
September 5, 2018	13:34	3A	route from Breitenauer See, start section	Training
September 5, 2018	-	4A	route from Breitenauer See, end section	Training
August 17, 2018	08:06	B	round trip from Vaihingen to Ditzingen	Training
August 17, 2018	08:47	B	round trip from Vaihingen to Ditzingen	Training
August 21, 2018	11:42	B	route from Vaihingen to Leonberg	Training
August 21, 2018	12:34	B	route from Vaihingen to Leonberg	Validation

present in the available vehicle data.

Once a valid trip is identified, the Route Analyzer application is used to request a GPS route match from the HERE REST API. Route Analyzer then matches the returned route information from HERE with the recorded vehicle data in time. The desired discretization window as described in Section 4.2.1 is shifted across the combined data creating the temporal training samples. As this is done, a B-spline is fitted to the respective prediction horizon of the output signal and the knots are saved to be used for training. Once complete, the position of each training sample along the route is determined and the corresponding spatial discretization of surrounding route information is created and saved.

Finally, any scaling of the data is done according to Table 4.1 and then split into the different road classes. It is saved in a compact form and is then ready for training. It is important to note that the training data uses the fitted B-spline knots instead of the true recorded data when calculating the error. This means that errors in the B-spline fit will be present when training the network. However, these errors are considered negligible and

acceptable for the work being done.

4.4.2 Training

All training of the networks is conducted using a standard laptop computer with a 2.7GHz Intel i7 quad-core processor equipped with an Nvidia GPU with 4 GB GDDR5 VRAM. The optimizer utilized for performing the gradient descent operation during training is *Adam*[49], an optimizer that has gained much popularity recently in training neural networks. It is a first-order method that uses estimates of the 1st and 2nd order moments of the gradients to determine the proper gradient step to apply to each weight.

A learning rate of $\gamma = 0.002$ is used during training since a low learning rate is typically used when using a stochastic gradient descent method on convolutional networks. All gradient steps are clipped using the gradient norm to prevent gradient vanishing and exploding throughout the multiple layers of the TCN residual blocks[54]. The clipping value is set to 1.0 after some minor experimentation. Other parameters for the optimizer are left as the default settings as found in [49].

Due to the length of training time for the large number of samples as each trip is discretized at 0.01 second intervals, only between 11 and 14 epochs are run for each road class network. One network is stopped after 8 epochs due to the obvious occurrence of over-fitting. In other words, the validation loss increases while the training loss continues to decrease. The tracked losses during training are the mean squared error (MSE) values across all samples for one epoch using the loss function outlined in Chapter 3.2.2. It is clear from the training results seen in Figure 4.6 that this is an acceptable training period and further training will only result in over-fitting.

It is clear from Figure 4.6 that the network for road class 1 performs very well with both low training and validation losses as compared to the other networks, however, the validation loss plateaus immediately after beginning training. The road class 5 network displays the worst case of over fitting in that it has a very large difference between validation

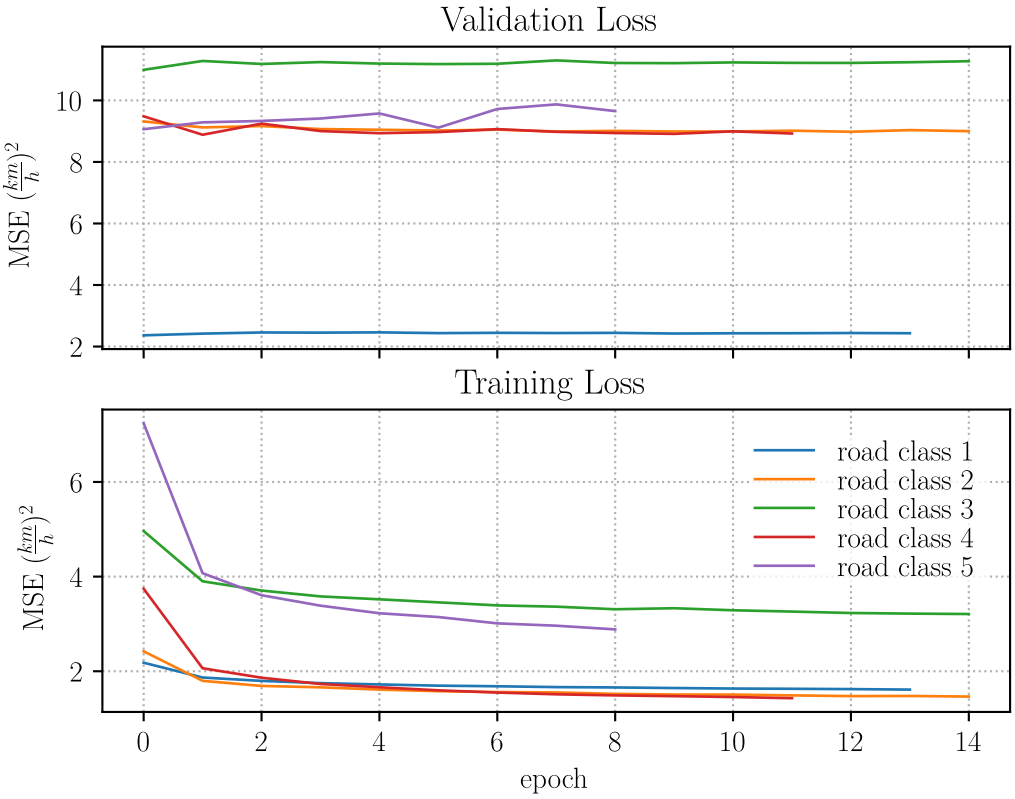


Figure 4.6: Training and validation losses for the full network.

and training losses and the validation loss increases noticeably throughout the training. This makes sense as there is not a large portion of the training data that consists of this road class and this road class is most likely to encounter unique, unpredictable situations such as interactions with pedestrians.

The temporal-only network was split by road class and trained the same way for 10 epochs. The results are shown in Figure 4.7. The training characteristics are very similar to those of the full network, however, the losses in general are much higher. The most notable difference, is that the road class 3 achieves a much worse fit for the training set than the full network. This road class contains the most stop and go events due to events like traffic signals, which the temporal-only network does not have as inputs.

While it is clear to see that all road class networks experience some difficulty generalizing past the first epoch, more insight into the performance is gained when looking closer

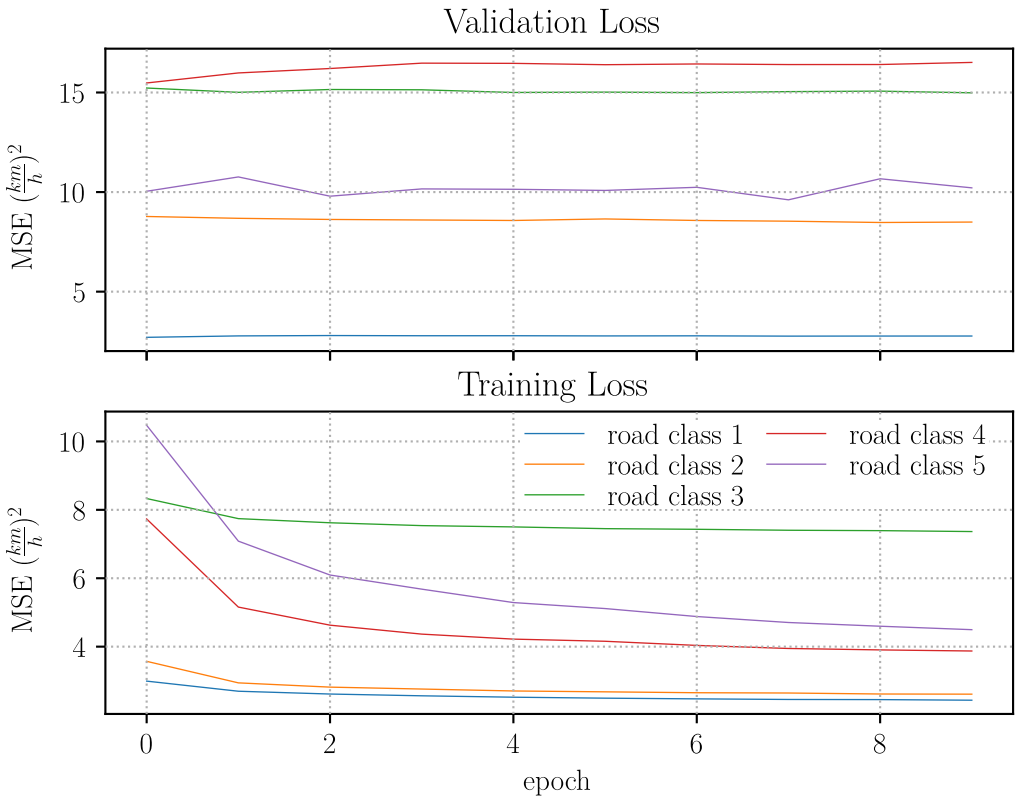


Figure 4.7: Training and validation losses for the temporal-only network.

at actual network predictions. This is done next, along with a comparison to two simple baseline prediction methods.

4.5 Prediction Results

For the results presented in this section, the different road class networks are considered as a single system and presented as such. When necessary, road class effects will be noted. For comparison purposes, two additional standard prediction methods are presented. These will help to provide a baseline for evaluating the proposed prediction method.

4.5.1 Baseline Prediction Methods

The two prediction methods to be used as a baseline are constant speed (CS) and constant acceleration (CA) as are also used in [15] for this purpose. The CS method simply assumes

the vehicle will not alter its speed during the prediction horizon. More specifically,

$$v[\mathrm{i}] = v[0] \quad \text{for i} = 0, 1, \ldots, T, \tag{4.9}$$

where $v[0]$ is the current speed and T is the number of future time steps to predict. This method is useful for very short prediction horizons and particularly in vehicle cruising conditions, such as on the highway (road class 1). The second method, CA, makes the assumption that the vehicle, or driver, will continue with the current maneuver for the prediction horizon by holding the longitudinal acceleration of the vehicle constant. This is given simply by

$$v[\mathrm{i}] = v[0] + a\mathrm{i}\Delta t_p \tag{4.10}$$

where Δt_p is the prediction time step size and a is obtained as

$$a = \frac{v[0] - v(-\Delta t_s)}{\Delta t_s}. \tag{4.11}$$

In this work, $\Delta t_s = 10\Delta t_p$ (0.1 seconds) as that is the sampling rate of the vehicle speed on the CAN bus. This method is more accurate than CS for a longer horizon and is more applicable for dynamic vehicle conditions.

4.5.2 Prediction Errors

Speed predictions were made every 0.01 seconds for each route using CS, CA, and the two proposed neural network approaches. The mean absolute error (MAE) of each prediction is calculated at 1 second intervals for the length of the prediction horizon: 5 seconds. The MAE is calculated for n predictions in each route using

$$\text{MAE} = \frac{1}{n}\sum_{\mathrm{j}=1}^{n} |v_{\text{prediction}} - v_{\text{measured}}|. \tag{4.12}$$

First, the prediction results for the TSNN and TONN are listed in Table 4.3 and the

Table 4.3: MAE at discrete prediction intervals.

	1 sec ($^{km}/_{h}$)		2 sec ($^{km}/_{h}$)		3 sec ($^{km}/_{h}$)		4 sec ($^{km}/_{h}$)		5 sec ($^{km}/_{h}$)	
	TSNN	TONN	TSNN	TONN	TSNN	TONN	TSNN	TONN	TSNN	TONN
Route 1A	**0.74**	0.89	**1.34**	1.62	**1.87**	2.32	**2.34**	2.99	**2.92**	3.66
Route 2A*	**0.83**	0.90	**1.67**	1.75	**2.52**	2.60	**3.33**	3.40	**4.10**	4.15
Route 3A	**0.55**	0.78	**1.00**	1.47	**1.39**	2.15	**1.75**	2.81	**2.19**	3.45
Route 4A	**0.58**	0.72	**1.14**	1.38	**1.71**	2.04	**2.27**	2.67	**2.84**	3.28
Route B 8/17-08:06	**0.69**	0.94	**1.25**	1.80	**1.78**	2.70	**2.29**	3.60	**2.88**	4.49
Route B 8/17-08:47	**0.58**	0.82	**1.11**	1.58	**1.64**	2.38	**2.15**	3.16	**2.67**	3.91
Route B 8/21-11:42	**0.66**	0.93	**1.19**	1.73	**1.68**	2.60	**2.15**	3.48	**2.72**	4.34
Route B* 8/21-12:34	**0.87**	1.11	**1.69**	2.13	**2.55**	3.20	**3.41**	4.23	**4.26**	5.22

* Validation route

lowest error of the prediction methods is in bold for each prediction interval. It is clear to see that the TSNN out performs the TONN in every prediction interval for each route. For this reason, the TONN results are omitted from inclusion in Table 4.4, where the errors of the TSNN and baseline prediction methods are listed. A graphical representation of all prediction method MAEs can be found in Figure 4.8. These results resemble the same trends found in [15]. It is clear to see that the CA prediction has the lowest error for the shortest prediction interval of 1 second, however, the TSNN MAE is very similar. Beyond that, the proposed TSNN method has the lowest error of the three methods as seen in Table 4.4. The exception to this is Route 2A, which is used as a validation route in training, so the TSNN has not ”seen” this data previously. It should be noted, however, that Route 2A is the reverse of Route 3A, which the network was trained on. Ultimately, at the longest prediction interval, the NN has the lowest MAE for every route.

A more intuitive graphical representation of the results can be seen in Figure 4.9 which shows the TSNN predictions made every 2 seconds along Route 2A versus the measured

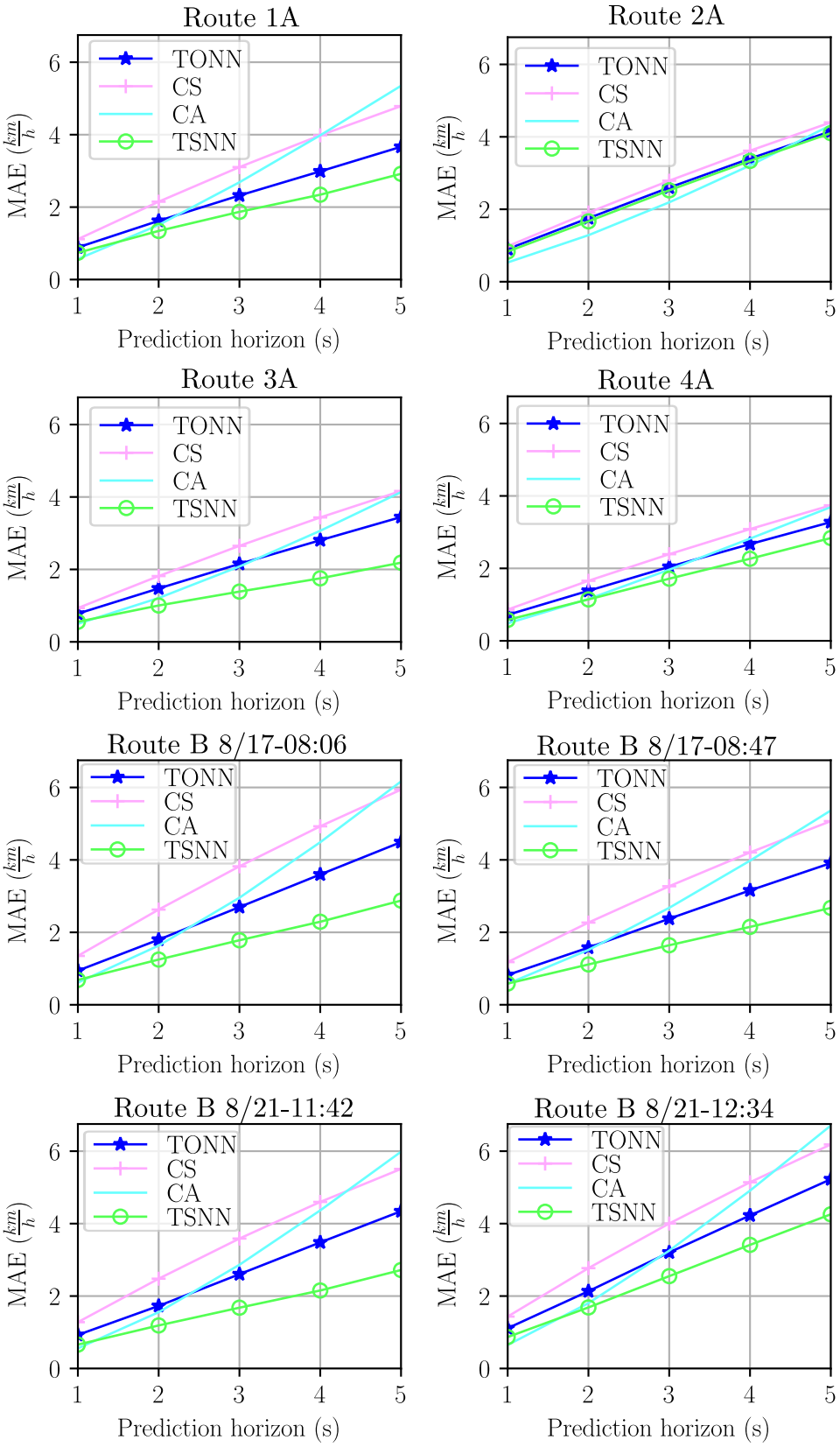


Figure 4.8: MAE for each route along the prediction horizon.

Table 4.4: MAE at discrete prediction intervals.

	1 sec ($^{km}/_{h}$)			2 sec ($^{km}/_{h}$)			3 sec ($^{km}/_{h}$)			4 sec ($^{km}/_{h}$)			5 sec ($^{km}/_{h}$)		
	CS	CA	NN	CS	CA	NN	CS	CA	NN	CS	CA	NN	CS	CA	NN
Route 1A	1.11	**0.57**	0.74	2.15	1.51	**1.34**	3.10	2.69	**1.87**	3.98	3.98	**2.34**	4.79	5.35	**2.92**
Route 2A*	0.98	**0.53**	0.83	1.91	**1.28**	1.67	2.79	**2.19**	2.52	3.62	**3.21**	3.33	4.40	4.32	**4.10**
Route 3A	0.93	**0.50**	0.55	1.82	1.21	**1.00**	2.65	2.08	**1.39**	3.43	3.07	**1.75**	4.17	4.13	**2.19**
Route 4A	0.86	**0.48**	0.58	1.66	1.16	**1.14**	2.39	1.96	**1.71**	3.08	2.82	**2.27**	3.74	3.69	**2.84**
Route B 8/17-08:06	1.35	**0.61**	0.69	2.63	1.63	**1.25**	3.82	2.96	**1.78**	4.93	4.49	**2.29**	5.94	6.17	**2.88**
Route B 8/17-08:47	1.18	**0.58**	0.58	2.27	1.51	**1.11**	3.27	2.68	**1.64**	4.20	3.98	**2.15**	5.07	5.36	**2.67**
Route B 8/21-11:42	1.28	**0.56**	0.66	2.48	1.55	**1.19**	3.59	2.86	**1.68**	4.60	4.37	**2.15**	5.51	5.98	**2.72**
Route B* 8/21-12:34	1.43	**0.65**	0.87	2.77	1.80	**1.69**	4.00	3.25	**2.55**	5.14	4.91	**3.41**	6.19	6.71	**4.26**

* Validation route
NOTE: TSNN is defined here as "NN" in this table

speed along the route. Similar plots for all routes can be found in Appendix B. The errors for each of the three prediction methods are also shown for the shortest and longest prediction interval. The displayed errors in Figure 4.9 are simply the difference between the predicted value and the measured value instead of the absolute error. It is presented this way, instead of using the absolute error, to show any patterns of bias in the prediction methods.

From Figure 4.9 it is possible to see that the CS and TSNN methods perform better at the 1 second prediction than the CA during the highway portion of the route (approx. 2100s - 3500s) despite the CA's overall better MAE. Beyond this section, however, there are large jumps in the errors of the CS and TSNN predictions, which account for the MAE results. This latter route section can be seen in Figure 4.10.

Upon closer examination of Figure 4.10 it is possible to identify the difficulties that the TSNN experiences when making predictions that explain the overall high error. In

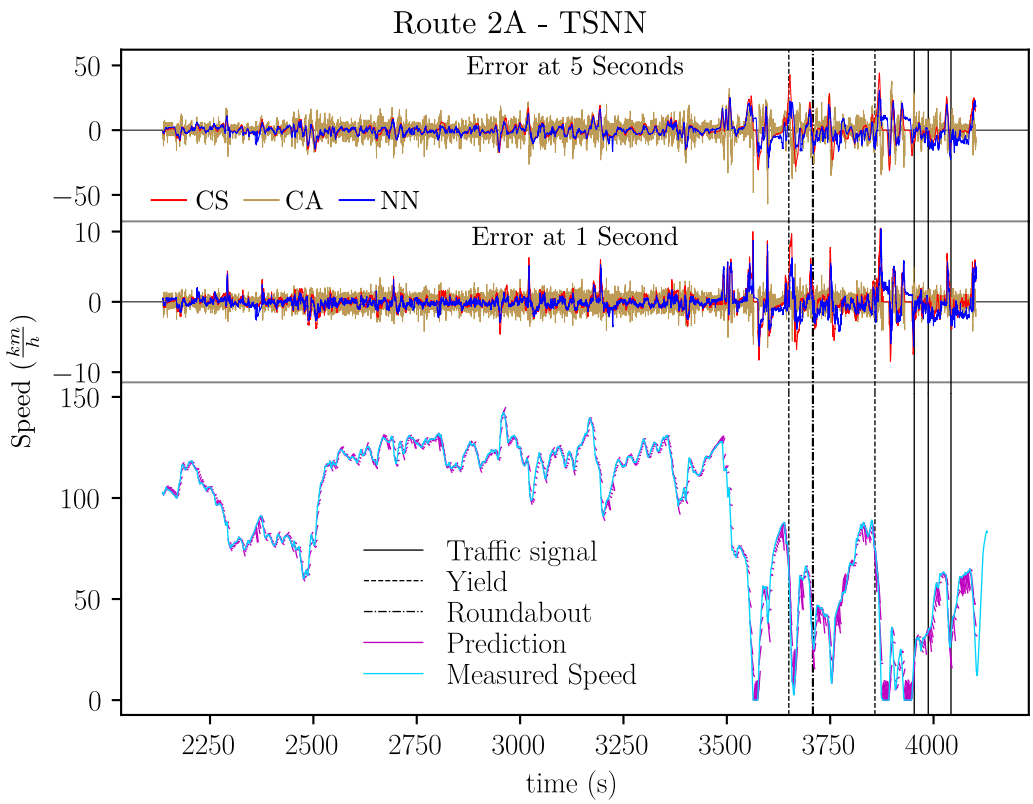


Figure 4.9: Speed Prediction and Errors for Route 2A, complete.

general, the TSNN is able to correctly predict how the speed profile will change in the near future in regards to speeding up or slowing down, but can vary drastically in the magnitude of this value. Specifically, it does very well in predicting the speed profile across the various route features indicated in Figure 4.10 as vertical lines. This is not surprising as the expected result for most driving scenarios in these locations will have the vehicle slow as it approaches the obstacle and then accelerate again, which the TSNN has properly identified. Under these conditions of large acceleration, even though the TSNN properly predicts the speed profile, it is still possible to see spikes in the error as it does not take much variation to result in a large speed deviation.

As expected, the largest additions to the overall error come from situations that cannot be predicted from the provided information given to the neural network. The most prevalent of these is seen every time the vehicle comes to a stop and the velocity reaches 0. The

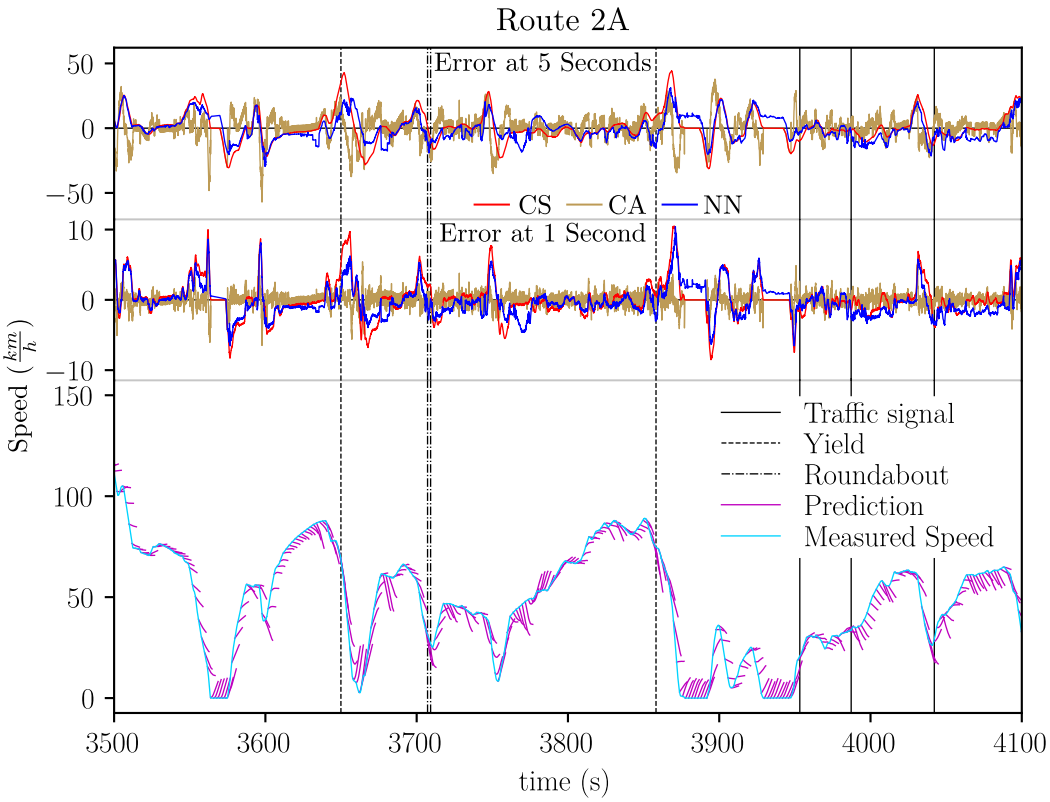


Figure 4.10: Speed Prediction and Errors for Route 2A, 3500s to 4100s.

TSNN continuously predicts the acceleration event that will eventually come, but it has no way of knowing exactly when.

4.6 Discussion of Results

From reviewing the results it is apparent that the TSNN prediction method was able to successfully identify speed profiles using the inclusion of knowledge of EGO location along the route. When compared to the TONN, it is made clear that the inclusion of HERE route information is what allows the TSNN to achieve such results. However, the exact magnitude of the speed predictions varied widely when compared between the training and validation sets resulting in less accurate predictions on the validation routes. This would be improved most effectively through the inclusion of more training data using different routes. As seen in Figure 4.6, the NN clearly had issues with generalization as the loss on

the validation set remained stagnant across all training epochs. This presented a difficulty in finding an appropriate network depth to capture fine effects on the speed, such as tip-in and tip-out of the accelerator pedal, without leading the network to simply memorize the training routes.

Speed predictions made by the TSNN were overall better than the baseline CS and CA methods with exception of the 1 second prediction horizon. At 1 second the CA predictions had the lowest error. This is likely due to immediate vehicle dynamics contributing more to the resulting speed than current or future effects of the driver, which is information that the TSNN is intended to include. This is directly seen when the vehicle is at rest and the TSNN continually predicts the start of motion, while the CA will continually predict the vehicle to stay at rest. When the vehicle is at rest for longer than 5 seconds, the overall error for the event will be larger for the TSNN.

A reduction in the overall prediction error of the TSNN is possible through a simple implementation of a heuristic to ignore the TSNN prediction when the vehicle is at rest. Alternatively, assuming after enough training data is used and the network is made deep enough, the TSNN prediction could hold for a shorter horizon since it should infer a speed increase from events such as a preceding vehicle moving or the brake pedal being released and then be increased again to the full length horizon once the vehicle is moving.

Overall, due to an unforeseen difficulty in finding and extracting appropriate vehicle test drives from the available database, an ultimately insufficient training set was used leading to the presented results. Better results may be obtained from performing dedicated test driving experiments to obtain training data, however, it is recommended that further work should focus first on optimizing the extraction of the already available vehicle test data. An increase in accuracy may also be obtained from pruning the training data to not include a sample every 0.01 seconds as there is likely many nearly identical samples when this is done leading to a bias in training.

CHAPTER 5
CONCLUSION

Many current approaches to improving energy efficiency of EVs rely on an accurate estimate of the future speed of the EGO vehicle. Many of these approaches utilize algorithms such as MPC to optimize the power output of the vehicle and require accurate short horizon speed prediction estimates in the range of 1 to 5 seconds. This work presented a new prediction method for use with NNs and the inclusion of spatial route information with temporal vehicle data to achieve the desired speed predictions.

The novel prediction method uses NNs to output control knots for a B-spline and uses the resulting B-spline as part of the loss function during training of the network. This method was demonstrated in the use of identifying the controller and motor dynamics of an EV with an LSTM NN. The B-spline prediction method proved unable to capture the fine dynamics of the system, likely due to the filtering effects of using too few B-spline knots. The method, however, was shown able to match the slower dynamics of the system with reasonable accuracy with no fine tuning of parameters. It was then chosen for use with the speed prediction as there would not be such characteristics in the speed output signal.

A new parallel-input TSNN was proposed to combine temporal and spatial information of the EGO vehicle at positions along a determined route to predict vehicle speed. A comparison of an identical network structure without including the spatial information showed that there is a clear advantage for the NN prediction when spatial route information is present. A comparison with two simple baseline prediction methods was also presented and found the TSNN achieved overall lower prediction errors. Despite these results, the TSNN was unable to capture many driving effects and showed signs of over fitting due to an insufficient amount of training data.

It is recommended that initial future efforts focus on the extraction and preparation

of existing vehicle data that has already been obtained from test vehicles. With too little training data, it is not possible to train a network deep enough to capture the effects shown in Chapter 4 without over fitting the data.

Including a metric to account specifically for individual driver behaviors such as those presented in [55] is also very likely to improve the predictions.

Further work can be done to validate the effectiveness of the presented B-spline prediction method. An investigation into knot selection versus accuracy and training times would provide helpful insight to when this method is applicable and when it should be avoided. A direct comparison with comparable parametric models could further prove the effectiveness of this method.

As with all NNs, there exist many hyper parameters that can drastically affect the performance. Future studies may include varying parameters such as network depths, activation functions, and learning rates.

Finally, the implementation of an online version of the proposed method would not require much additional work and would allow for the integration and testing with existing efficiency improving controllers.

APPENDIX A

NN MODEL SUMMARIES

A.1 Motor Identification NN

```
Model: "LSTM_network"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 256)               270336
_________________________________________________________________
dropout (Dropout)            (None, 256)               0
_________________________________________________________________
activation (Activation)      (None, 256)               0
_________________________________________________________________
dense (Dense)                (None, 256)               65792
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
activation_1 (Activation)    (None, 256)               0
_________________________________________________________________
knots_output (Dense)         (None, 9)                 2313
=================================================================
Total params: 338,441
Trainable params: 338,441
Non-trainable params: 0
_________________________________________________________________
```

A.2 TSNN

```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
temporal_input (InputLayer)     [(None, 50, 7)]      0
__________________________________________________________________________________________________
spatial_input (InputLayer)      [(None, 45, 9)]      0
__________________________________________________________________________________________________
tcn (TCN)                       (None, 16)           17456       temporal_input[0][0]
__________________________________________________________________________________________________
conv1d (Conv1D)                 (None, 42, 32)       1184        spatial_input[0][0]
__________________________________________________________________________________________________
dropout (Dropout)               (None, 16)           0           tcn[0][0]
__________________________________________________________________________________________________
activation_1 (Activation)       (None, 42, 32)       0           conv1d[0][0]
__________________________________________________________________________________________________
```

activation (Activation)	(None, 16)	0	dropout[0][0]
flatten (Flatten)	(None, 1344)	0	activation_1[0][0]
tf_op_layer_strided_slice (Tens	[(None, 50)]	0	temporal_input[0][0]
concatenate (Concatenate)	(None, 1410)	0	activation[0][0] flatten[0][0] tf_op_layer_strided_slice[0][0]
dropout_1 (Dropout)	(None, 1410)	0	concatenate[0][0]
dense (Dense)	(None, 256)	361216	dropout_1[0][0]
activation_2 (Activation)	(None, 256)	0	dense[0][0]
dropout_2 (Dropout)	(None, 256)	0	activation_2[0][0]
dense_1 (Dense)	(None, 256)	65792	dropout_2[0][0]
activation_3 (Activation)	(None, 256)	0	dense_1[0][0]
knots_output (Dense)	(None, 10)	2570	activation_3[0][0]

Total params: 448,218
Trainable params: 448,218
Non-trainable params: 0

A.3 TONN

Layer (type)	Output Shape	Param #	Connected to
temporal_input (InputLayer)	[(None, 101, 7)]	0	
tcn (TCN)	(None, 16)	17456	temporal_input[0][0]
dropout (Dropout)	(None, 16)	0	tcn[0][0]
activation (Activation)	(None, 16)	0	dropout[0][0]
tf_op_layer_strided_slice (Tens	(None, 101)	0	temporal_input[0][0]
concatenate (Concatenate)	(None, 117)	0	activation[0][0] tf_op_layer_strided_slice[0][0]
dropout_1 (Dropout)	(None, 117)	0	concatenate[0][0]
dense (Dense)	(None, 256)	30208	dropout_1[0][0]
activation_1 (Activation)	(None, 256)	0	dense[0][0]

```
dropout_2 (Dropout)             (None, 256)          0           activation_1[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                 (None, 256)          65792       dropout_2[0][0]
____________________________________________________________________________________________________
activation_2 (Activation)       (None, 256)          0           dense_1[0][0]
____________________________________________________________________________________________________
knots_output (Dense)            (None, 10)           2570        activation_2[0][0]
====================================================================================================
Total params: 116,026
Trainable params: 116,026
Non-trainable params: 0
____________________________________________________________________________________________________
```

APPENDIX B

ROUTE SPEED PREDICTIONS

The prediction results for all routes for both the TSNN network and TONN network from Chapter 4 are included in the following pages.

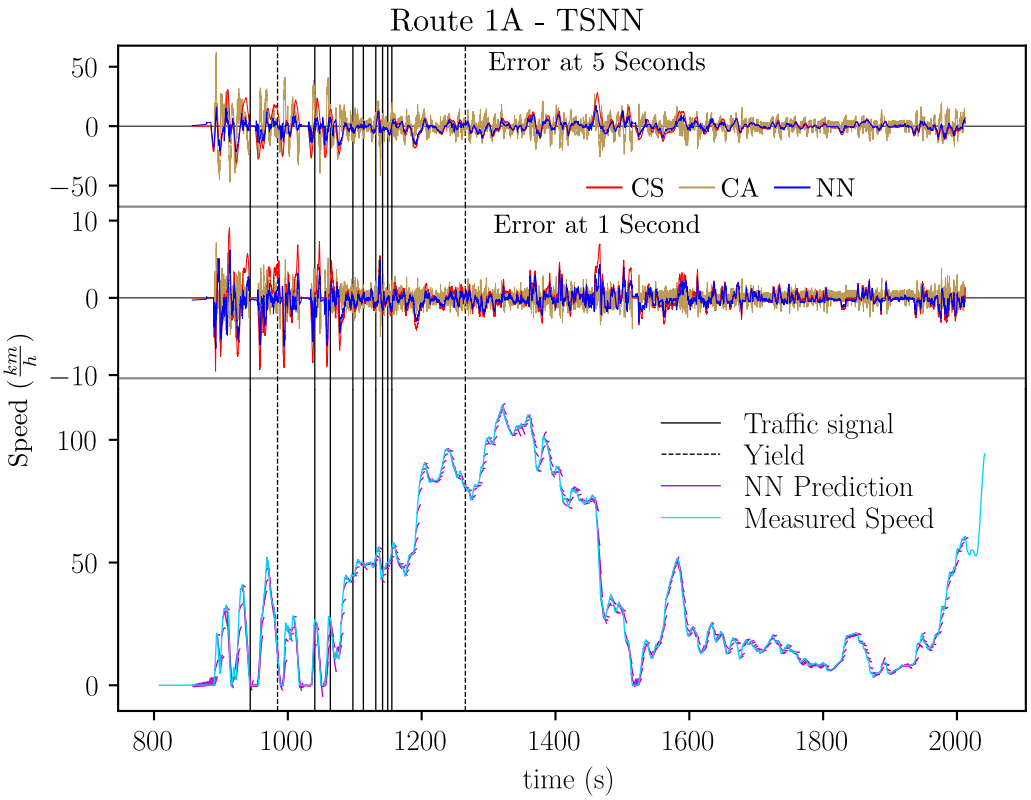


Figure B.1: Speed Prediction and Errors for Route 1A with TSNN.

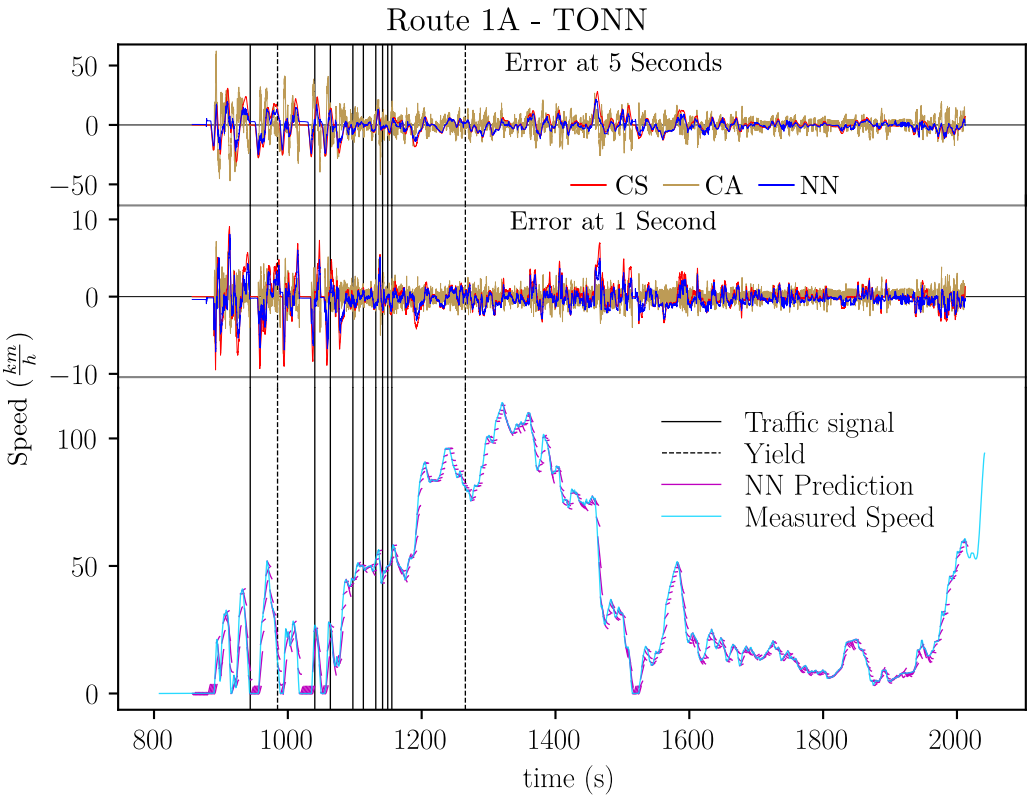


Figure B.2: Speed Prediction and Errors for Route 1A with TONN.

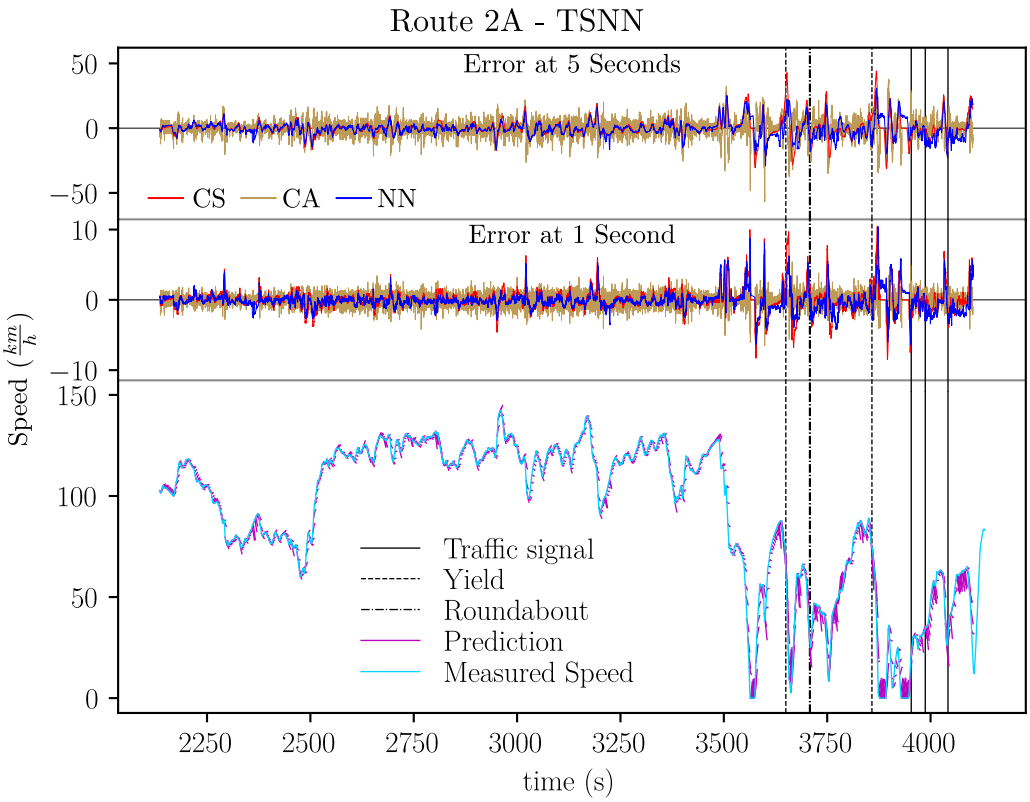


Figure B.3: Speed Prediction and Errors for Route 2A with TSNN.

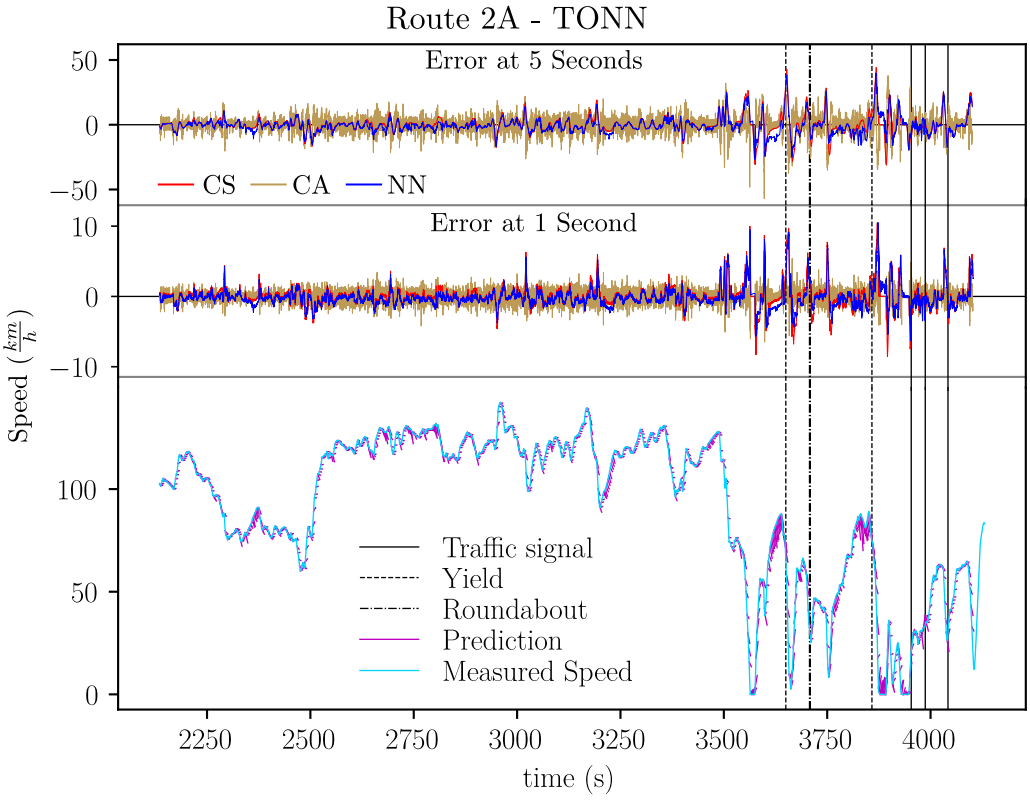


Figure B.4: Speed Prediction and Errors for Route 2A with TONN.

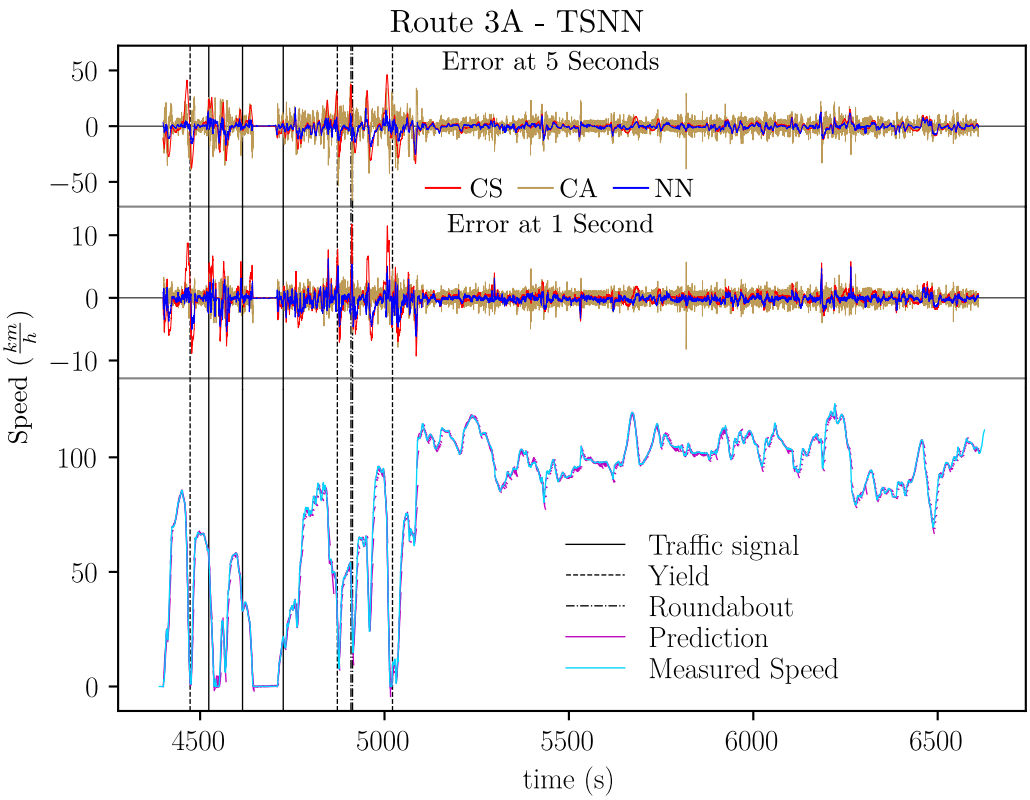


Figure B.5: Speed Prediction and Errors for Route 3A with TSNN.

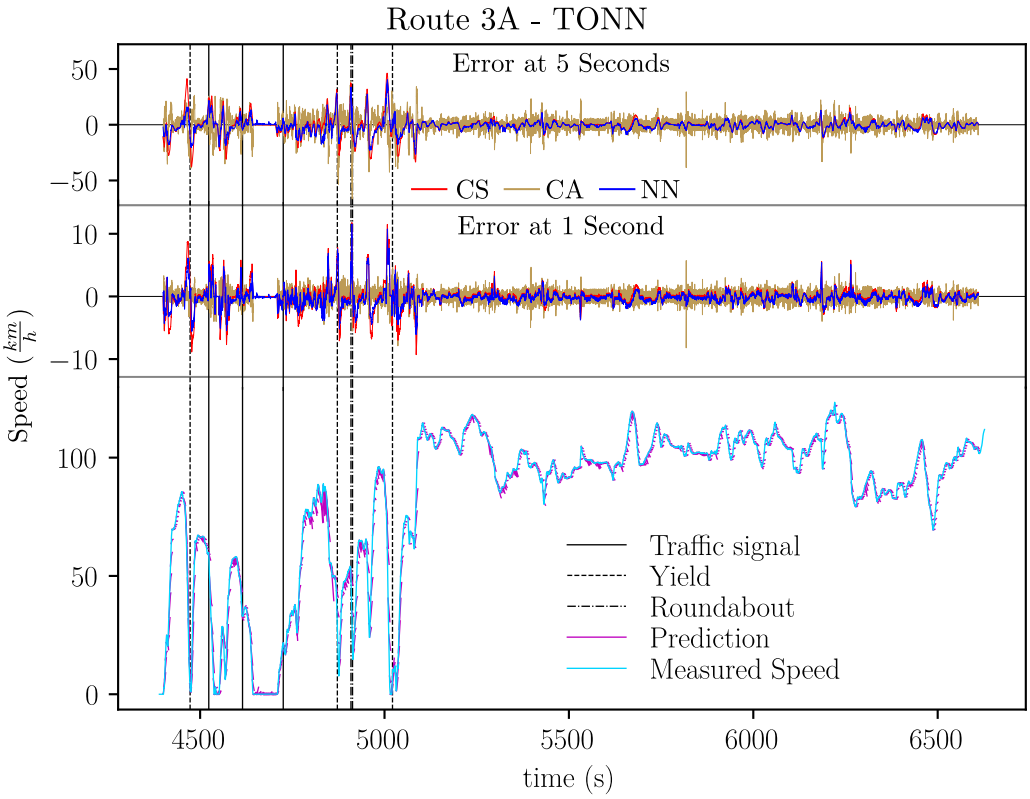


Figure B.6: Speed Prediction and Errors for Route 3A with TONN.

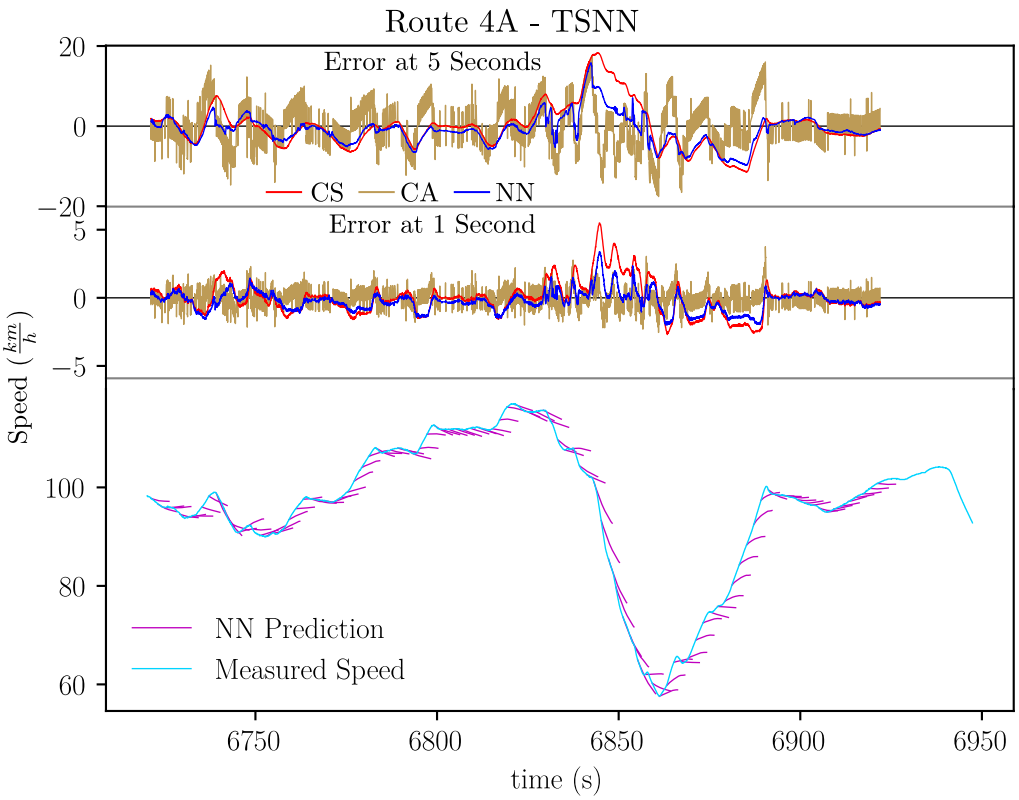


Figure B.7: Speed Prediction and Errors for Route 4A with TSNN.

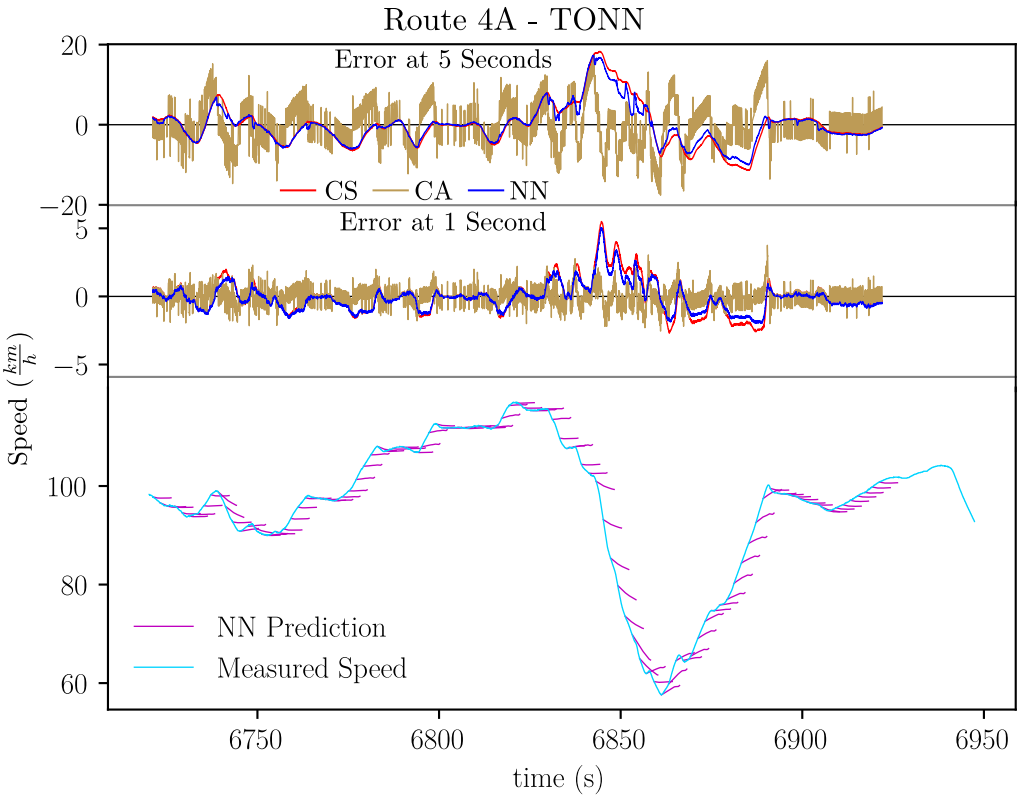


Figure B.8: Speed Prediction and Errors for Route 4A with TONN.

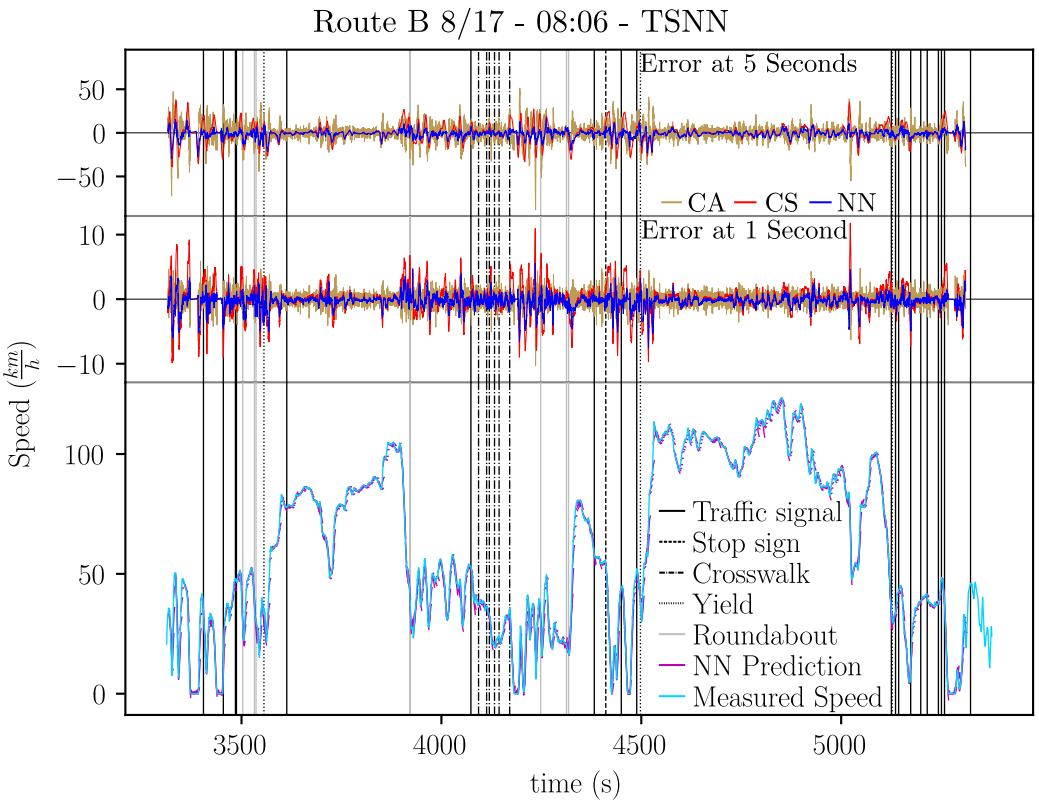


Figure B.9: Speed Prediction and Errors for Route B on 8/17 at 08:06 with TSNN.

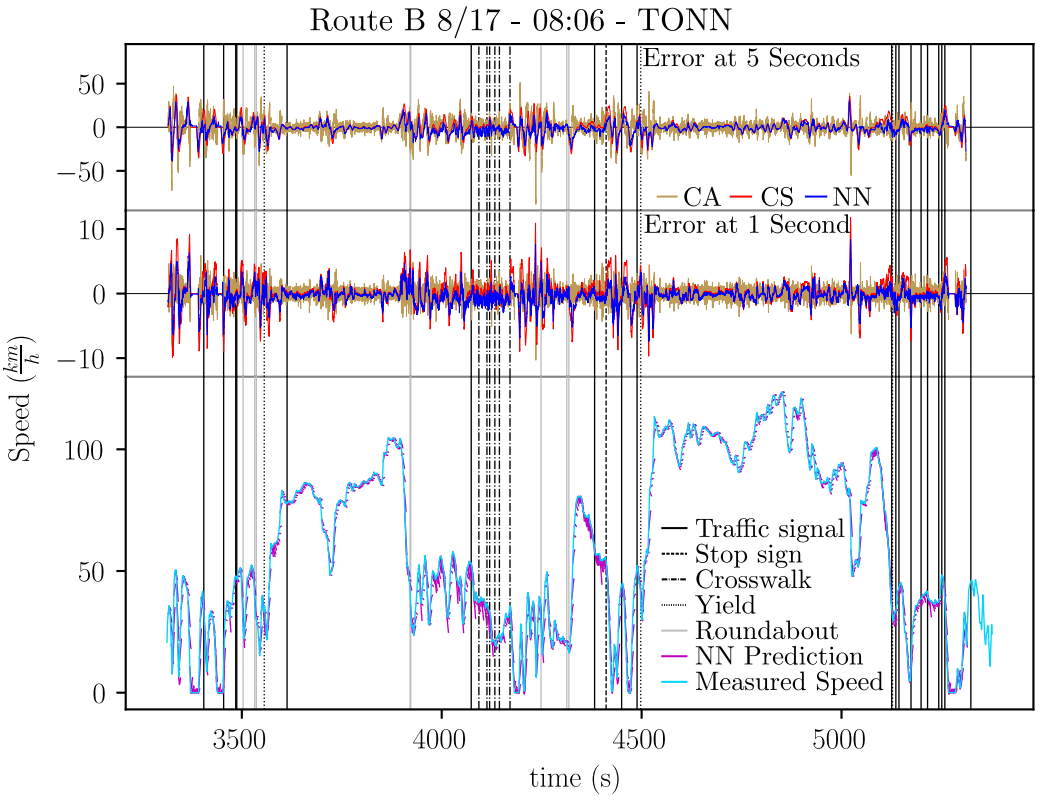


Figure B.10: Speed Prediction and Errors for Route B on 8/17 at 08:06 with TONN.

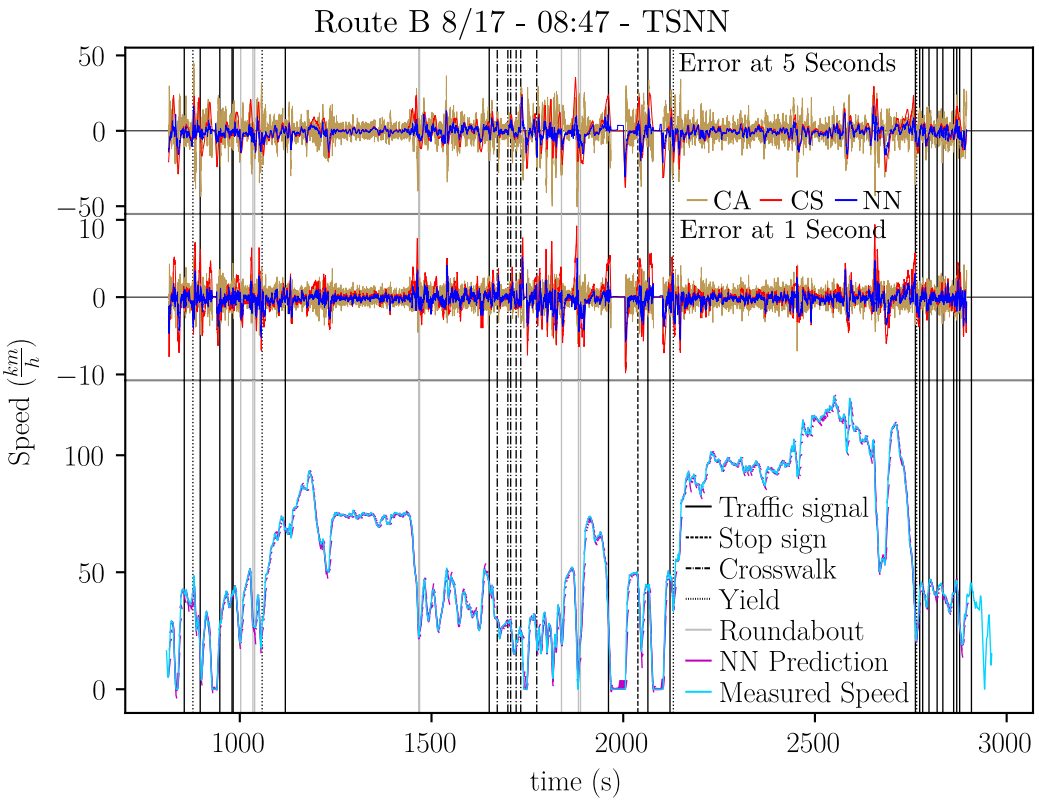


Figure B.11: Speed Prediction and Errors for Route B on 8/17 at 08:47 with TSNN.

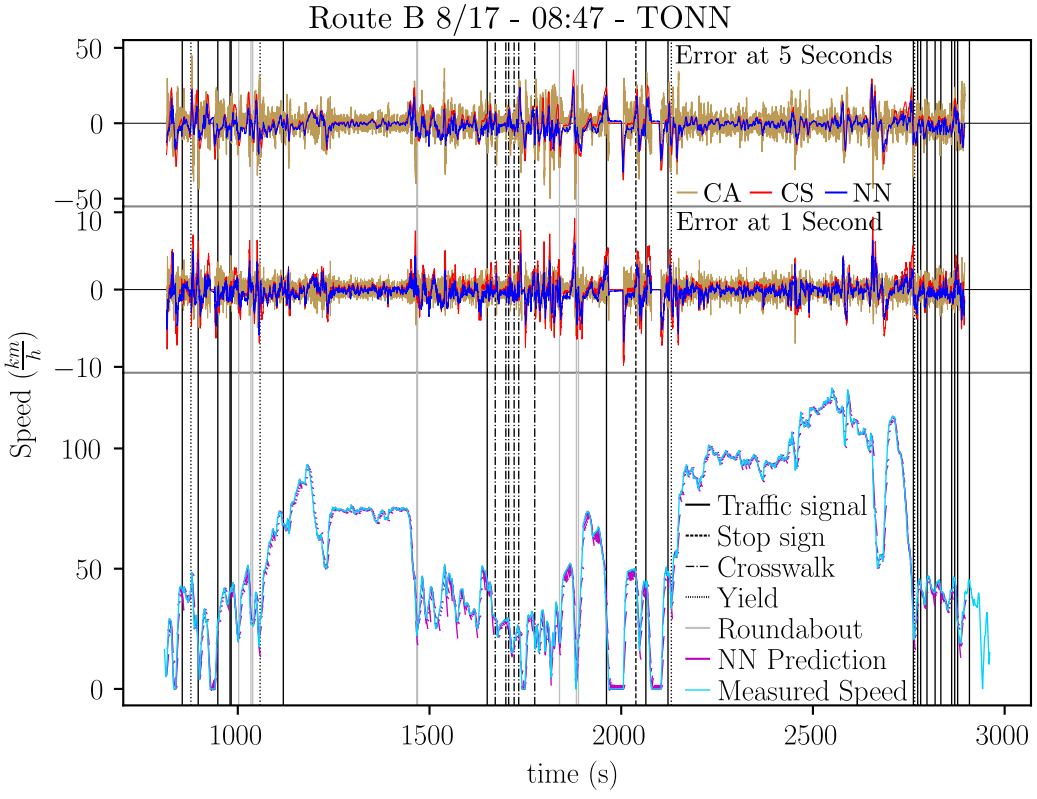


Figure B.12: Speed Prediction and Errors for Route B on 8/17 at 08:47 with TONN.

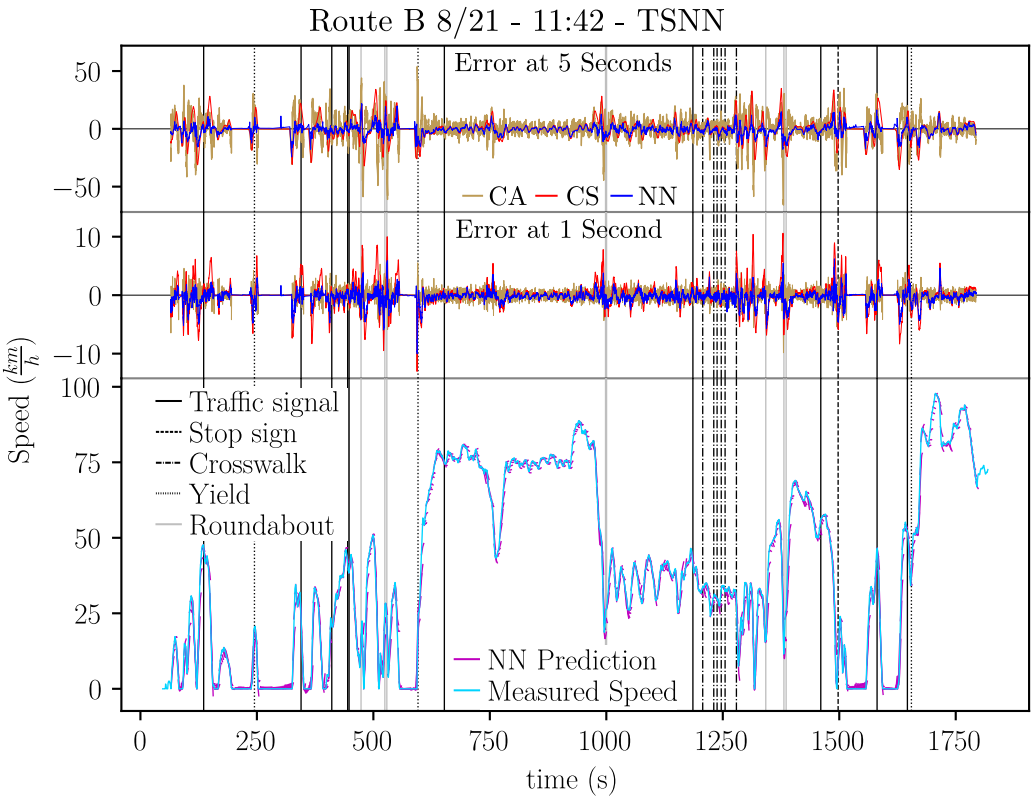


Figure B.13: Speed Prediction and Errors for Route B on 8/21 at 11:42 with TSNN.

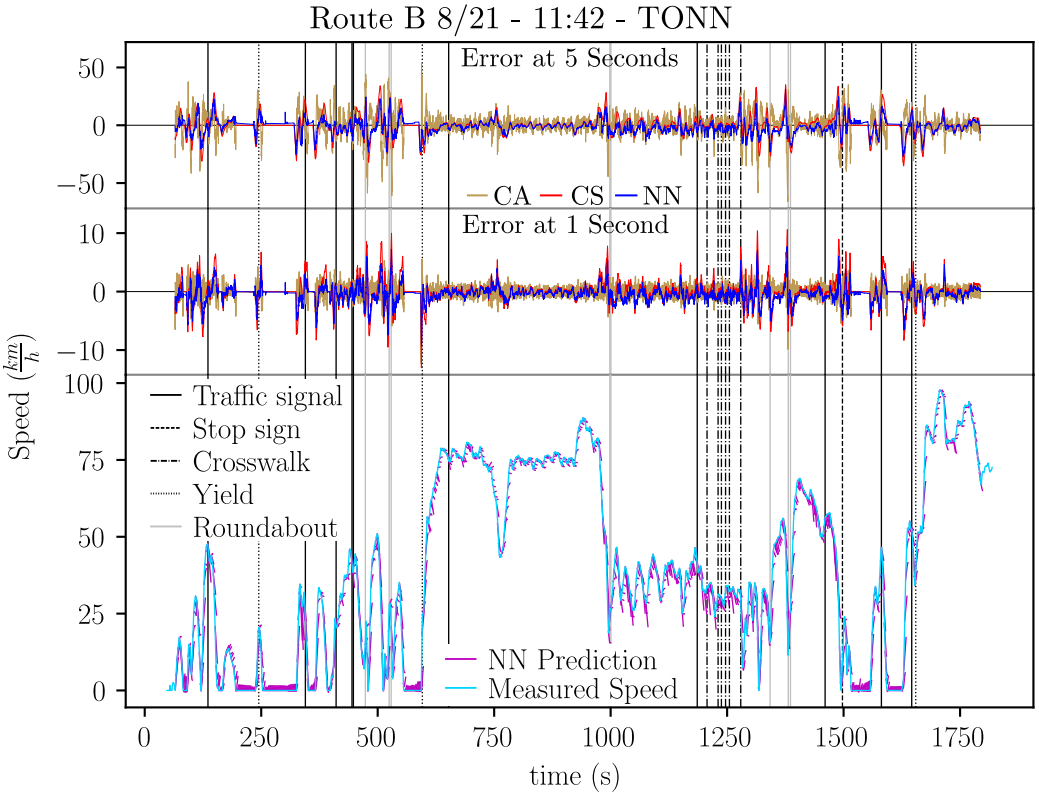


Figure B.14: Speed Prediction and Errors for Route B on 8/21 at 11:42 with TONN.

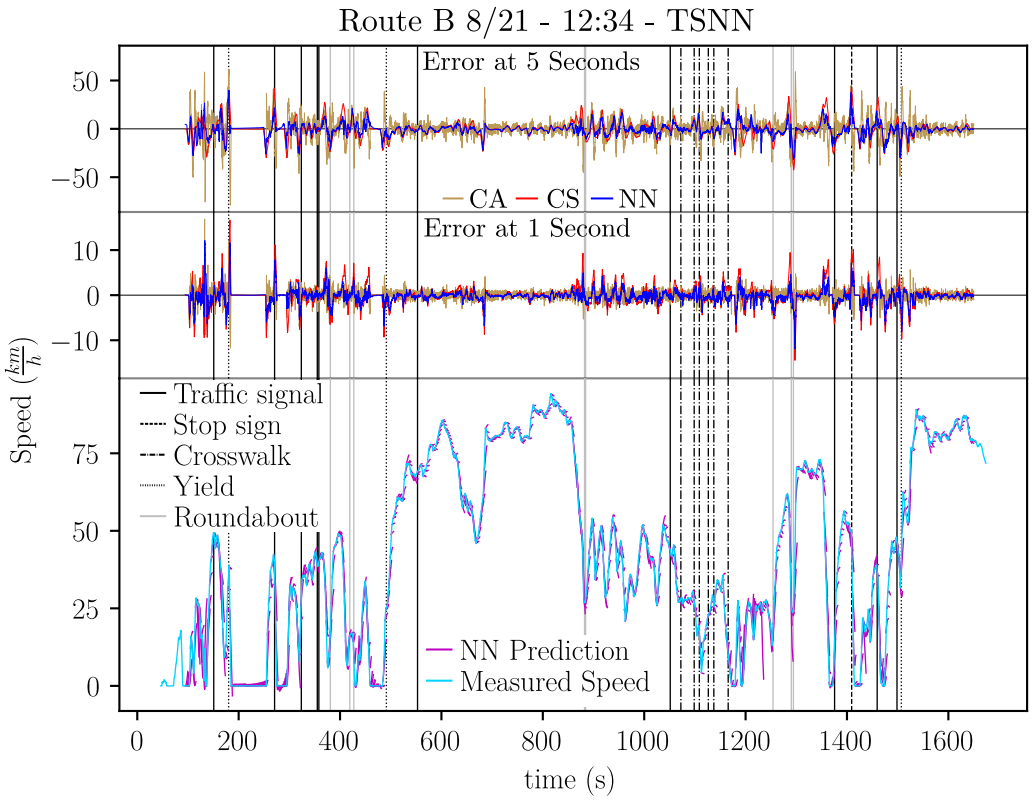


Figure B.15: Speed Prediction and Errors for Route B on 8/21 at 12:34 with TSNN.

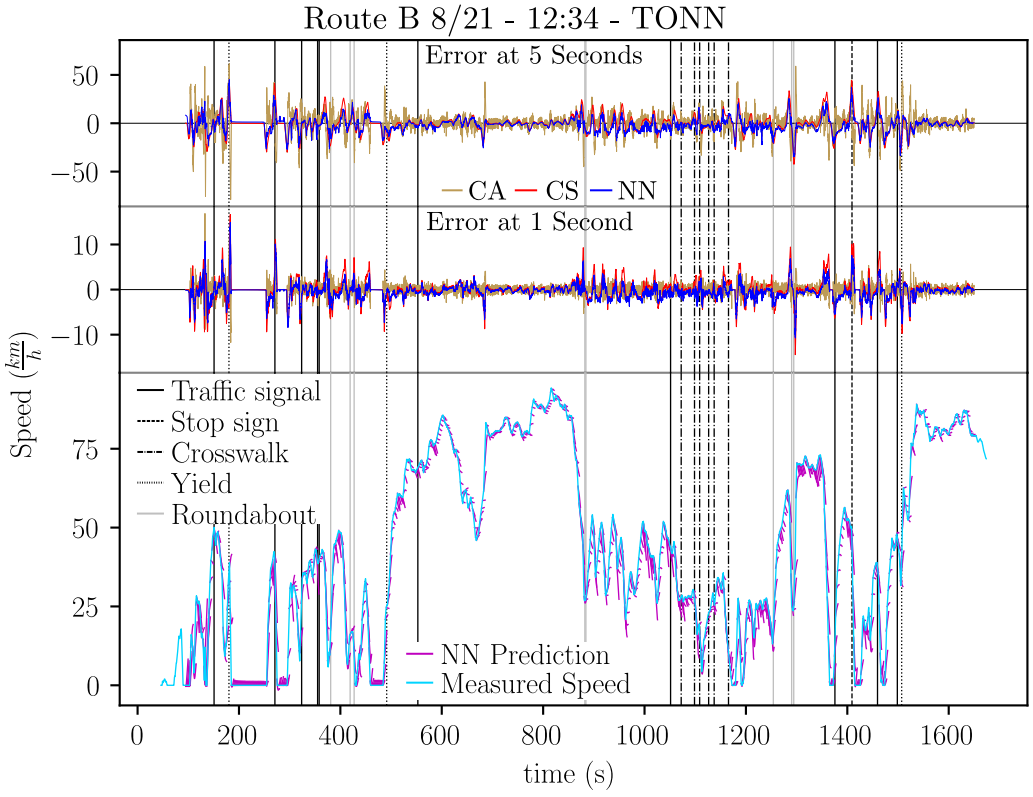


Figure B.16: Speed Prediction and Errors for Route B on 8/21 at 12:34 with TONN.

REFERENCES

[1] T. Chen and H. Chen, "Approximation capability to functions of several variables, nonlinear functionals, and operators by radial basis function neural networks," *IEEE Transactions on Neural Networks*, vol. 6, no. 4, pp. 904–910, 1995.

[2] W. Baxt, "Application of artificial neural networks to clinical medicine," *The Lancet*, vol. 346, no. 8983, pp. 1135 –1138, 1995.

[3] M. Schrage and D. Kiron. (Sep. 28, 2018). Machine learning in the automotive industry: Aligning investments and incentives. M. S. M. Review, Ed.

[4] A. Akula and S. R. Devi, "Automatic speed-limit sign detection and recognition for advanced driver assistance systems," *International Journal of Innovative Technology and Exploring Engineering*, B. E.I.E. S. Publication, Ed., 2019.

[5] M. Johanson, S. Belenki, J. Jalminger, M. Fant, and M. Gjertz, "Big automotive data: Leveraging large volumes of data for knowledge-driven product development," in *Proc. of the IEEE International Conference on Big Data*, 2014, pp. 736–741.

[6] J. Miller, L. Du, and D. Kodjak, "Impacts of world-class vehicle efficiency and emissions regulations in select g20 countries," in *Proc. of the G20 Transport Task Group*, I. C. C. Transp., Ed., 2017.

[7] J. Saeed, M. Niakinezhad, N. Fernando, and L. Wang, "Model predictive control of an electric vehicle motor drive integrated battery charger," in *IEEE 13th International Conference on Compatibility, Power Electronics and Power Engineering*, 2019, pp. 1–6.

[8] X. Lin, M. Hu, S. Song, and Y. Yang, "Battery-supercapacitor electric vehicles energy management using dp based predictive control algorithm," in *Proc. of the IEEE Symposium on Computational Intelligence in Vehicles and Transportation Systems*, 2014, pp. 30–35.

[9] D. Liu, Y. Wang, X. Zhou, and Z. Lv, "Extended range electric vehicle control strategy design and muti-objective optimization by genetic algorithm," in *Proc. of the Chinese Automation Congress*, 2013, pp. 11–16.

[10] S. Glaser, O. Orfila, L. Nouveliere, R. Potarusov, S. Akhegaonkar, F. Holzmann, and V. Scheuch, "Smart and green acc, adaptation of the acc strategy for electric vehicle with regenerative capacity," in *Proc. of the IEEE Intelligent Vehicles Symposium*, 2013, pp. 970–975.

[11] F. Ye, P. Hao, X. Qi, G. Wu, K. Boriboonsomsin, and M. J. Barth, "Prediction-based eco-approach and departure at signalized intersections with speed forecasting on preceding vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 4, pp. 1378–1389, 2019.

[12] D. Wu, Y. Li, C. Du, H. Ding, Y. Li, X. Yang, and X. Lu, "Fast velocity trajectory planning and control algorithm of intelligent 4wd electric vehicle for energy saving using time-based mpc," *IET Intelligent Transport Systems*, vol. 13, no. 1, pp. 153–159, 2019.

[13] F. Borrelli, A. Bemporad, and M. Morari, *Predictive control for linear and hybrid systems*. C, 2017.

[14] A. Rezaei and J. B. Burl. (2015). Prediction of vehicle velocity for model predictive control. IFAC-PapersOnLine, Ed.

[15] S. Lefèvre, C. Sun, R. Bajcsy, and C. Laugier, "Comparison of parametric and non-parametric approaches for vehicle speed prediction," in *Proc. of the American Control Conference*, 2014, pp. 3494–3499.

[16] Y. Liu, Y. Wang, X. Yang, and L. Zhang, "Short-term travel time prediction by deep learning: A comparison of different lstm-dnn models," in *Proc. of the IEEE 20th International Conference on Intelligent Transportation Systems*, 2017, pp. 1–8.

[17] B. Jiang and Y. Fei, "Vehicle speed prediction by two-level data driven models in vehicular networks," *IEEE Transactions on Intelligent Transportation Systems*, vol. 18, no. 7, pp. 1793–1801, 2017.

[18] J. Lemieux and Y. Ma, "Vehicle speed prediction using deep learning," in *IEEE Vehicle Power and Propulsion Conference (VPPC)*, 2015, pp. 1–5.

[19] Z. Cheng, M. Chow, D. Jung, and J. Jeon, "A big data based deep learning approach for vehicle speed prediction," in *Proc. of the IEEE 26th International Symposium on Industrial Electronics*, 2017, pp. 389–394.

[20] F. Abas, M. Morteza, and J. Mostafa, "Vehicle's velocity time series prediction using neural network," *International Journal of Automotive Engineering*, 2011.

[21] J. Park, Y. Murphey, J. Kristinsson, R. McGee, M. Kuang, and T. Phillips, "Intelligent speed profile prediction on urban traffic networks with machine learning," in *Proc. of the International Joint Conference on Neural Networks*, 2013, pp. 1–7.

[22] F. Morlock, B. Rolle, M. Bauer, and O. Sawodny, "Time optimal routing of electric vehicles under consideration of available charging infrastructure and a detailed con-

sumption model," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–13, 2019.

[23] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," in *Bulletin of Mathematical Biophysics*, G. Palm and A. Aertsen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1943, pp. 229–230, ISBN: 978-3-642-70911-1.

[24] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, pp. 65–386, 1958.

[25] P. Werbos and P. John, "Beyond regression : New tools for prediction and analysis in the behavioral sciences /," Jan. 1974.

[26] S. Boyd and L. Vandenberghe, *Convex Optimization*. USA: Cambridge University Press, 2004, ISBN: 0521833787.

[27] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, J. A. Anderson and E. Rosenfeld, Eds., pp. 696–699, 1986.

[28] P. J. Werbos, "Backpropagation through time: What it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[29] I. Sutskever, "Training recurrent neural networks," AAINS22066, PhD thesis, Toronto, Ont., Canada, 2013, ISBN: 978-0-499-22066-0.

[30] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," *Technische Universität München*, Apr. 1991.

[31] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997.

[32] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, Sep. 1989.

[33] M. A. Nielsen, *Neural networks and deep learning*, misc, 2018.

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 9, pp. 1904–1916, 2015.

[35] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *ArXiv*, vol. abs/1803.01271, 2018.

[36] Hampshire and Waibel, "A novel objective function for improved phoneme recognition using time delay neural networks," in *Proc. of the International Joint Conference on Neural Networks*, 1989, 235–241 vol.1.

[37] A. Pandey and D. Wang, "Tcnn: Temporal convolutional neural network for real-time speech enhancement in the time domain," in *Proc. of the IEEE International Conference on Acoustics, Speech and Signal Processing*, 2019, pp. 6875–6879.

[38] G. Zhou and F. Chen, "Urban congestion areas prediction by combining knowledge graph and deep spatio-temporal convolutional neural network," in *Proc. of the 4th International Conference on Electromechanical Control Technology and Transportation*, 2019, pp. 105–108.

[39] E. Shelhamer, J. Long, and T. Darrell, "Fully convolutional networks for semantic segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 4, pp. 640–651, 2017.

[40] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, 2015. arXiv: 1512.03385 [cs.CV].

[41] T. Salimans and D. P. Kingma, *Weight normalization: A simple reparameterization to accelerate training of deep neural networks*, 2016. arXiv: 1602.07868 [cs.LG].

[42] S. Loffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015. arXiv: 1502.03167 [cs.LG].

[43] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. of the Fourteenth International Conference on Artificial Intelligence and Statistics*, G. Gordon, D. Dunson, and M. Dudík, Eds., ser. Proceedings of Machine Learning Research, vol. 15, Fort Lauderdale, FL, USA: PMLR, 2011, pp. 315–323.

[44] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[45] N. Passalis, A. Tefas, J. Kanniainen, M. Gabbouj, and A. Iosifidis, "Deep adaptive input normalization for price forecasting using limit order book data," *ArXiv*, vol. abs/1902.07892, 2019.

[46] L. B. Rall, *Automatic Differentiation: Techniques and Applications*. 1981.

[47] C. Boor, *A Practical Guide to Splines*. Jan. 1978, vol. Volume 27.

[48] T. Lyche and K. Morken, *Spline Methods*. Jan. 5, 2005.

[49] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. arXiv: 1412.6980 [cs.LG].

[50] Z. Zhang, Y. Asakawa, T. Imamura, and T. Miyake, "Experiment design for measuring driver reaction time in driving situation," in *Proc. of the IEEE International Conference on Systems, Man, and Cybernetics*, 2013, pp. 3699–3703.

[51] J. Khashbat, T. Tsevegjav, J. Myagmarjav, I. Bazarragchaa, A. Erdenetuya, and N. Munkhzul, "Determining the driver's reaction time in the stationary and real-life environments (comparative study)," in *Proc. of the 7th International Forum on Strategic Technology*, 2012, pp. 1–3.

[52] B. Shi, L. Xu, J. Hu, Y. Tang, H. Jiang, W. Meng, and H. Liu, "Evaluating driving styles by normalizing driving behavior based on personalized driver modeling," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 45, no. 12, pp. 1502–1508, 2015.

[53] E. Hellström and M. Jankovic, "A driver model for velocity tracking with look-ahead," in *Proc. of the American Control Conference*, 2015, pp. 3342–3347.

[54] R. Pascanu, T. Mikolov, and Y. Bengio, *On the difficulty of training recurrent neural networks*, 2012. arXiv: 1211.5063 [cs.LG].

[55] T. Waters, "Adaptive driver modeling using machine learning algorithms for the energy optimal planning of velocity trajectories for electric vehicles and realizing simultaneous lane keeping and adaptive speed regulation on accessible mobile robot testbeds," Master's thesis, Georgia Institute of Technology, Jan. 9, 2018.